INTELLIGENCE AS A FEATURE: MODELING ML IN SOFTWARE PRODUCT LINES

Luz-Viviana Cobaleda [1], Andrés López [1,2], Paola Vallejo [3], Raúl Mazo [2], Julián Carvajal [1]

[1] Facultad de Ingeniería, Universidad de Antioquia, Medellín, Colombia.
[2] Lab-STICC, ENSTA, Brest, Francia.
[3] Escuela de Ciencias Aplicadas e Ingeniería, Universidad EAFIT, Medellín, Colombia.

ABSTRACT

The integration of Machine Learning (ML) components into modern software systems enhances data-driven decision-making but introduces new challenges for Software Product Line (SPL) engineering. Variability modeling, configuration, and reuse become increasingly complex when adaptive ML components are involved. Although previous studies have addressed variability in traditional SPLs and ML integration in standalone systems, limited work has systematically explored the intersection of these two domains. This paper presents a structured framework that extends SPL engineering to support ML-aware variability management. The framework enables the systematic modeling and configuration of ML components and has been implemented in the VariaMos web tool. A case study demonstrates the framework's feasibility and applicability, illustrating how it supports the development of adaptive and intelligent product lines.

KEYWORDS

Machine Learning (ML), Software Product Lines (SPL), ML-based systems, variability modeling.

1. INTRODUCTION

The rapid evolution of artificial intelligence (AI) over the last decade stems from advances in computational power, the availability of massive datasets, and increasingly sophisticated algorithms. As a result, AI has become a transformative technological force, empowering software-intensive systems with new capabilities across diverse domains [1], [2], [3], [4]. AI-based systems are essentially software systems whose functionalities are enabled by at least one AI component (e.g., for image and speech recognition or autonomous driving) [4]. However, incorporating AI components into software products introduces new software engineering challenges and amplifies existing ones. The situation becomes even more critical when these components are integrated not only into a single product but into a family of software products or a Software Product Line (SPL). Thus, the integration of Machine Learning (ML) components into SPLs introduces new dimensions of variability that traditional modeling techniques are not prepared to handle. This raises fundamental questions: How can an AI/ML component be modeled within an SPL? How can architects effectively integrate ML components into their SPLs? What information about the model is necessary to enable a successful SPL configuration process? The inability of current modeling approaches to address these questions reveals a significant research gap. Additionally, the integration of ML components into software systems introduces unique challenges that have given rise to the field of Software Engineering for AI (SE4AI). Recent literature has systematically identified the issues that emerge across the software lifecycle, impacting areas such as requirements engineering, architecture, testing, deployment,

DOI: 10.5121/ijsea.2025.16601

and maintenance [3], [4], [5], [6]. While these challenges are broad, this paper focuses on those most relevant to the design of SPL.

Most research in SE4AI has focused on the challenges of integrating ML components into individual software systems. In the context of SPLs, where systematic reuse is central, these challenges persist but evolve into variability management problems. For instance, defining performance metrics for a single product is an engineering task, whereas managing multiple components with diverse performance profiles across products becomes a variability challenge. However, literature explicitly addressing this transformation of ML-related challenges in SPLs remains scarce, revealing a significant research gap. Among the documented issues in individual systems, requirements engineering is particularly critical: both customers[3] and development teams [6] often overestimate ML capabilities, leading to unrealistic expectations such as perfect accuracy or zero false positives[4]. This gap between business goals and technical specifications is compounded by the dynamic nature of ML components, which introduces new and still poorly understood quality attributes—such as freshness and robustness[3], [4] —and trade-offs, including fairness versus accuracy [3], [4].

Although these challenges are significant for individual systems, their impact is amplified in SPLs, where systematic reuse and variability management are essential. The inclusion of ML components introduces additional variability concerns—such as defining performance metrics at the product line level, aligning stakeholder understanding, and specifying monitoring policies—that extend beyond individual products. Despite extensive research on AI-related software components, the literature still lacks approaches that explicitly address their distinctive characteristics within SPLs[3], [4].

In this paper, we propose a framework for enhancing SPLs by considering intelligence as a first-class feature and enabling the seamless integration of ML components. The main contribution lies in a specification-oriented approach that systematically guides the integration of ML-based functionalities into SPLs, addressing key aspects such as variability management, probabilistic feature modeling, ML component characterization, continuous monitoring, component replacement, and product derivation with ML components. This framework promotes consistent reuse, customization, and traceability of intelligent features across product configurations within the SPL context.

This proposal builds upon and improves the version presented initially in [7] by providing an enhanced demonstration of the framework's feasibility and applicability through a running example and its implementation in the VariaMos web tool (www.variamos.com). As part of an ongoing effort to operationalize and validate its practical use, this web-based tool leverages a microservices architecture to support the specification of product lines through a **multi-language modeling approach**, as well as reasoning over products and product lines.

The remainder of the paper is structured as follows: Section 2 provides background information on SPL engineering and ML components documentation. Section 3 introduces the running example in the virtual store domain, which will be referenced throughout the remainder of the paper. Section 4 presents the proposed framework for designing SPLs with ML components and discusses the implications of this approach. Section 5 reviews related work. Finally, Section 6 concludes the paper and outlines directions for future research.

2. BACKGROUND

The design and development of SPLs rely on systematic approaches to manage variability and promote reuse across families of related software systems. To provide the necessary foundation for the proposed framework, this section outlines the core concepts of SPL engineering and the integration of ML components.

2.1. SPL and Variability Management

A SPL represents a systematic approach to developing families of related applications within a specific domain through strategic reuse of common assets [8]. This paradigm leverages shared components and systematic variability management to achieve significant reductions in development time and costs while improving product quality through the incorporation of proven, reusable artifacts.

Software Product Line Engineering (SPLE) operationalizes this approach through two fundamental processes, as presented in Figure 1: (1) Domain engineering, which establishes reusable assets and variability models, and (2) Application engineering, which derives specific products from these shared resources[8]. Variability—the capacity of a system to be adapted or configured for specific contexts—serves as the core mechanism enabling this systematic reuse across diverse product requirements.

1) Domain engineering establishes the foundation of reusable assets through two sequential phases. **A) Domain analysis** identifies and specifies SPL variability using formal models such as feature models [9], which define variation points, available alternatives, and constraint relationships. This phase encompasses: domain requirements definition to capture stakeholder needs and scope constraints, reference architecture specification aligned with domain requirements, and variability model quality assurance through systematic verification, diagnosis, and validation activities. **B) Domain implementation** transforms abstract specifications into concrete, reusable components. Key activities include requirements engineering for domain components, architectural design specification, domain component implementation, comprehensive unit testing, and explicit linkage between components and variability model elements. This phase produces the core asset base comprising domain components, architectural models, and associated test suites.

2) Application engineering derives specific products through the systematic configuration and instantiation of domain assets across two phases. **A) Configuration and customization management** captures customer-specific requirements and configures variability models accordingly, encompassing application requirements engineering, variability model configuration, application architecture definition, and component customization to meet specific product needs. **B) Derivation** constructs final products from configured domain assets through requirements engineering for the derivation process, assembly architecture definition, systematic product implementation from domain components, and comprehensive system integrity testing, including performance, validation, and audit verification.

This dual-process framework ensures systematic reuse while maintaining the flexibility necessary to address diverse product requirements within the target domain. The SPLE framework applies to various domains, including, but not limited to, education [10], agricultural systems [11], smart buildings [12], e-commerce [13], automotive manufacturing [14], and information systems [15]. Our running example belongs to the e-commerce domain.

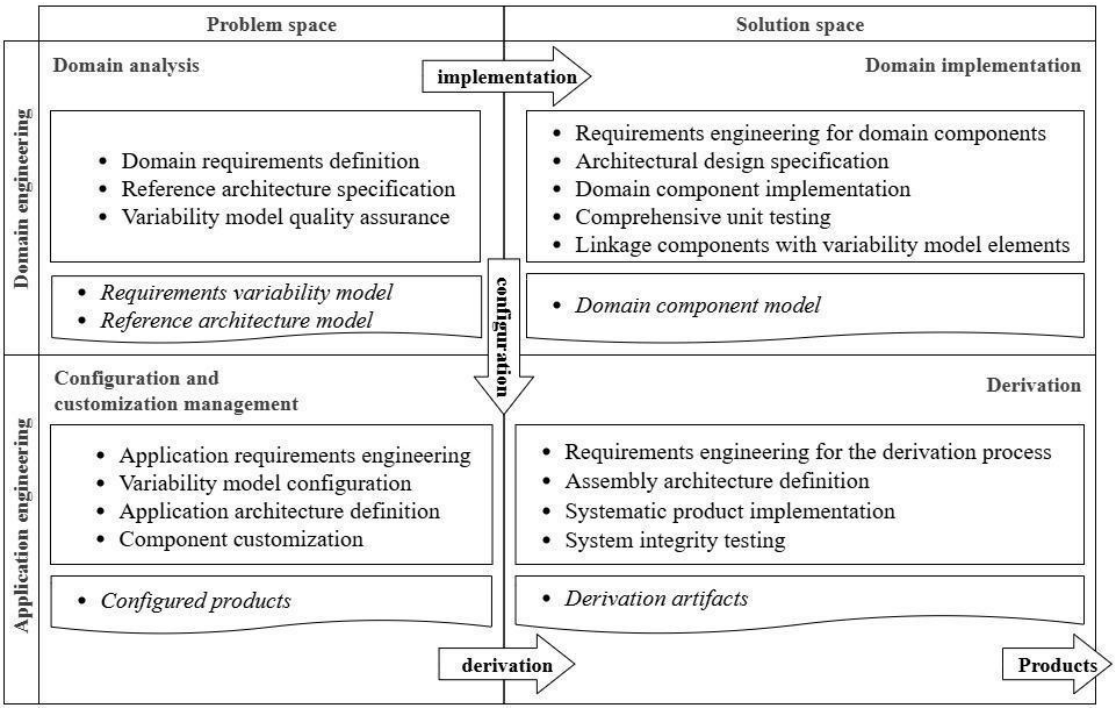

Figure 1. SPLE framework implemented in the VariaMos web tool, from [8]

2.2. ML Components

An **ML component** is a special type of software component that encapsulates ML models along with their associated data processing, inference logic, and system integration capabilities [6], [16]. These components constitute the primary means of integrating ML capabilities into complex software systems, acting as a bridge between the underlying ML models and the overall system architecture. Component reuse is a foundational principle that enables the efficient development of multiple products from a shared, common core.

3. RUNNING EXAMPLE: E-COMMERCE

To illustrate the applicability of the proposed Framework for ML-Aware Variability, we present a running example in the virtual store domain. Virtual stores constitute a representative and relevant domain for SPLs. These platforms enable the online exchange of goods and services, allowing businesses to publish product catalogs and customers to perform transactions. Although they may operate in diverse markets, such as fashion, electronics, or digital services, their primary goal is to facilitate efficient and secure commercial transactions among multiple users. Virtual stores thus represent a cornerstone of modern e-commerce systems.

Our SPL for virtual stores captures a set of core components common to most instances, including a Product catalog (for listing and managing items), a Shopping cart (to collect selected products before purchase), a Payment module (for processing transactions via various methods), and a Delivery system (to coordinate product shipment or digital access). Beyond these shared functionalities, the SPL supports a wide range of variability points. For instance, products may differ in terms of user authentication mechanisms, catalog presentation styles, supported payment gateways, shipping logistics, or user interface customization. In addition to these structural variations, ML components introduce a new layer of variability that enhances user experience and system efficiency. Examples include a semantic search engine (trained to interpret the

context of user queries), a sentiment analysis module (applied to customer reviews), and a content moderation system (which classifies and filters inappropriate content).

These ML-based components add complex dimensions of variability related to model architecture, inference thresholds, latency, and human-in-the-loop decision-making. For example, the content moderation module can be instantiated in two configurations: (i) a human-assisted moderation system, where a toxicity classification model flags suspicious comments for manual review, and (ii) an automated moderation system, where the same model flags comments for automatic censorship using a higher confidence threshold. Choosing between these configurations entails trade-offs. The automated system requires higher model precision to minimize false positives (*i.e.*, unjustified censorship). In contrast, the human-assisted configuration can operate with a lighter, lower-latency model, as human moderators make the final decisions. To implement such ML functionalities, the SPL can rely on pre-trained models (*e.g.*, text classifiers or Sentence Transformers) obtained from public repositories, such as Hugging Face. Within the SPL context, these models can be encapsulated as reusable components, enabling developers to integrate intelligent behavior without having to redesign models from scratch for each product instance. This running example will be referenced throughout the paper to demonstrate the modeling of variability, the configuration of intelligent components, and the systematic reuse of ML assets in SPL.

4. A FRAMEWORK FOR ML-AWARE VARIABILITY

The integration of ML components into SPL represents a fundamental paradigm shift that challenges the traditional assumptions underlying systematic software reuse. While conventional SPL approaches have proven effective for deterministic software components with predictable behavior and stable interfaces [17], [18], ML components introduce unprecedented complexity through their inherent stochasticity, data dependency, continuous evolution requirements, and non-functional characteristics that defy traditional software engineering practices [3], [4].

The proposed framework is organized into five interconnected phases that collectively address the complete lifecycle of ML-enhanced SPLs: **ML-aware domain analysis**, **Adaptive architecture design, ML-aware domainimplementation**, **Dynamic product configuration**, and **Product derivation and validation** of its resulting products. Each phase builds upon established SPL theory while introducing novel concepts and recommended practices specifically designed to handle the probabilistic nature, performance variability, and operational complexity inherent in ML systems.

4.1. ML-Aware Domain Analysis

The domain analysis phase requires significant adaptations when ML components are involved, particularly in feature modeling and architectural decision [8]. Traditional Boolean feature satisfaction proves inadequate for ML components whose capabilities vary across contexts and exhibit probabilistic behavior [6]. A key distinction of **ML-based features** lies in their reliance on **training data properties**. The performance and capabilities of these features are susceptible to the characteristics of the training data, including its quality, representativeness, and intrinsic attributes. Additionally, implementing ML-based features can introduce risks associated with sensitive data, particularly regarding privacy, security, and information governance, due to the implications of data use and storage for model training and inference.

Recommendation 1: Implement Probabilistic Feature Modeling.

SPL engineers should extend conventional feature models to capture the uncertainty in ML component capabilities [9]. Rather than relying on binary feature satisfaction, engineers should model features with quality distributions that reflect variability in ML component performance.

Practical Implementation: For each feature that will be satisfied by an ML component, SPL engineers should identify it as an "*ML-based feature*" and define the following Feature Quality Profile:

```
FeatureQualityProfile = {
feature_id: String,
feature_type: type,
ml_component_id: String,
quality_distribution: {
       accuracy_range: [min_accuracy, max_accuracy],
       context_sensitivity: Map[Context, AccuracyLevel],
       confidence_intervals: Map[Scenario, ConfidenceRange]
}}
```

Application in the E-commerce Example: In our running example, the fraud detection feature is defined with an accuracy ranging between 0.88 and 0.95. This metric is influenced by context—decreasing to 0.75 for international transactions during weekends, but reaching 0.98 for transactions originating from suspicious IP addresses. Additionally, the confidence intervals are established to classify the level of certainty of the machine learning model; for example, values below 0.70 can be considered low confidence, values between 0.70 and 0.84 are considered medium confidence, and values equal to or above 0.85 are considered high confidence. Figure 2 shows the implementation of the Probabilistic Feature Modeling in the VariaMos tool; this representation enables the management of the inherent uncertainty in ML-based features, supporting more accurate reasoning mechanisms and better-informed, adaptive, and reliable product configurations.

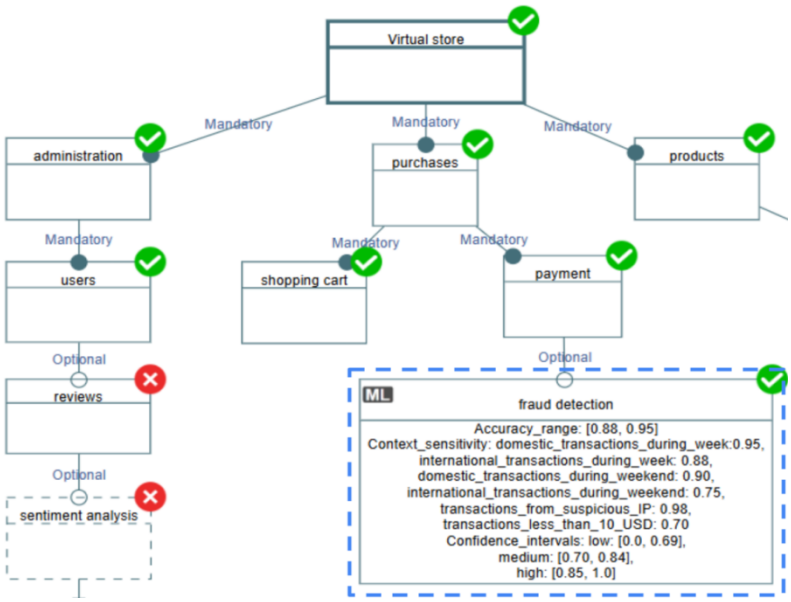

Figure 2. Probabilistic feature modeling implemented in the VariaMos web tool.

4.2. Adaptive Architecture Design

The reference architecture must explicitly address the dynamic and context-sensitive nature of ML components. ML models often evolve, depend on external data sources, and exhibit probabilistic behavior that affects system reliability and performance. Therefore, architectural decisions must incorporate design strategies that manage adaptability and traceability, ensure periodic updates, and maintain the long-term stability and performance of integrated ML functionalities. These strategies should align feature variability, model capabilities, and operational constraints, which is paramount for ensuring the robustness, adaptability, scalability, and maintainability of the ML-based SPL.

Recommendation 2: Design ML-Aware Reference Architecture.

The reference architecture must account for several key aspects.

- It must provide for a clear separation of concerns between the core SPL framework, the ML model development cycle, the deployment pipeline, and model monitoring.
- It must support various deployment strategies, including on-device (edge computing), on-premises, or cloud-based, depending on the specific product requirements and constraints.
- It must ensure data privacy, security, and compliance, while facilitating seamless integration with robust ML engineering practices, such as MLOps.

Practical Implementation: SPL engineers should be able to:

- Use microservice-based architecture, where ML components are deployed as decoupled services accessible through well-defined APIs.
- Use of containerization (*e.g.*, Docker) to package models and their dependencies, ensuring environmental consistency and portability.

4.3. ML-aware Domain Implementation

The domain implementation phase requires a structured approach to documenting, versioning, and managing ML components. This approach should be complemented by a formal monitoring process that can detect performance degradation and automatically trigger component replacement procedures. Effectively characterizing and selecting suitable ML components is essential to understanding their capabilities, limitations, and performance profiles. This enables successful integration and reduces associated risks. The monitoring system is designed to address the dynamic and non-deterministic nature of ML components by identifying potential degradation in the production environment and issuing alerts. Additionally, careful consideration is required for some aspects. For example, orchestrating ML components across products involves managing dependencies, activation conditions, and contextual adaptation. Furthermore, replacing ML components systematically requires mechanisms to evaluate, decouple, and reintegrate new versions with minimal disruption.

Recommendation 3: Adopt Intelligent Component Characterization.

To ensure the precise and systematic characterization of pre-trained ML components, it is proposed that **Model Cards** be mandatorily adopted. **Model Cards**, introduced by Mitchell [19] and further extended by Toma [20], provide a standardized framework for documenting ML models in a transparent and structured manner. This approach recommends customizing specific

sections of the standard Model Card, such as Model Details, Intended Use, SPL reusability Profile, Model Usage, Operational Requirements, Performance Metrics, and Caveats. These cards are tailored for domain experts who, while not data scientists, are responsible for selecting and integrating third-party components.

Practical Implementation: For each ML component in the SPL, a standardized model card is proposed, capturing the following essential information:

```
ModelCard= {
model_details: {
      model_id: String, version: ModelVersion,
      developed_by: String, model_type: MLModelType,
      license: LicenseSpecification
}, intended_use: {
      primary_use: String, out-of-scope_use: String
}, spl_reusability_profile: {
      supported_domains: Set[Domain],
      integration_complexity: String, (e.g., "Low")
}, model_usage: {
      api_endpoint: String,
      deployment_guidance: String
}, performance_metrics: Map[clave, valor],
   operational_requirements: {
            cpu: CPUSpecification, ram: RAMSize, gpu: String, notes: String
   },
   caveats: [String]
}
```

The SPL-aware Model Card specification defines the essential attributes for characterizing an ML component. The purpose and content of each key attribute are detailed below:

- **model_details**: Provides technical specifications—covering developer information, version control, model architecture, training methodology, and licensing terms that define commercial use rights, current license type, and redistribution permissions.
    - **model_id**: A unique identifier for the model, such as its name in a public repository.
    - **version**: The specific version of the model, following semantic versioning where possible, to track changes and dependencies.
    - **developed_by**: The organization, team, or individual responsible for the model's development.
    - **model_type**: Specifies the model's task category (*e.g.*, Text Classification, Object Detection), informing its functional role.
    - **license**: The legal specification governing the use, modification, and distribution of the model, crucial for commercial product derivation.

- **intended_use**: Defines appropriate use cases, target applications, and intended user populations by outlining usage scenarios, specifying primary and out-of-scope applications, detailing the model's adaptability, and highlighting its limitations and potential biases.
    - **primary_use**: A concise description of the model's main purpose and the scenarios where it is designed to be applied (*e.g.*, real-time fraud detection).
    - **out-of-scope_use**: Explicitly states the limitations and use cases for which the model has not been designed or validated, preventing misuse.

- **spl_reusability_profile**: A section dedicated to evaluating the ML component's fitness as a reusable asset within the SPL context. This is a key input for variability modeling.
    - **supported_domains**: A set of application domains where the model has demonstrated reliable performance, highlighting potential domain biases.
    - **integration_complexity**: A categorical rating (*e.g.*, "Low", "Medium", "High") that estimates the engineering effort needed to integrate the component, based on its dependencies and API.

- **Model_usage**: Offers guidance on model consumption through various interfaces (*e.g.*, UI, API) and outlines its compatibility with different deployment platforms and operating systems. It also provides guidance on optimizing performance and outlines deployment strategies for different environments, including local setups and cloud platforms.
    - **api_endpoint**: The URL or interface for sending inference requests.
    - **deployment_guidance**: A summary of instructions and best practices for deploying the model in different environments (*e.g.*, cloud, edge).

- **performance_metrics**: Comprehensive performance evaluation including accuracy measures, uncertainty quantification, and decision thresholds.

- **operational_requirements**: Provides system requirements and hardware recommendations to help users prepare for deploying or fine-tuning the model in their computing environment.
    - **cpu**: The recommended minimum specification for the CPU. This is critical for overall system performance and serves as the primary compute resource when no GPU is used.
    - **ram**: The recommended minimum system RAM. This memory is required to hold the operating system, host application, model dependencies, and the model itself before being loaded into specialized hardware.
    - **gpu**: Specify whether a GPU is required, as well as its minimum specifications.
    - **notes**: Provides additional qualitative context or performance tips.

- **caveats and recommendations**: Presents caveats and recommendations by assessing potential societal impacts, fairness considerations, and bias mitigation strategies, while also outlining behavioral limitations related to “Not Safe For Work” (NSFW) content, including explicit material, violence, or hate speech.

This information empowers the SPL architect to make a reasoned configuration decision: either accept a component with known limitations and plan for specific monitoring, or select an alternative component whose characteristics better align with the product being built. In addition, the systematic adoption of Model Cards represents a crucial step toward responsible ML deployment by enhancing transparency around model behavior and operational boundaries. By standardizing technical and ethical documentation practices, Model Cards enable stakeholders to evaluate and compare models using multidimensional criteria that extend beyond traditional performance metrics to encompass fairness, inclusivity, and equity considerations.

Application in the E-commerce Example: In the sentiment analysis case of our running example, we defined a domain component that integrates two ML components: the *DistilBERT*model from the Hugging Face repository (https://huggingface.co/distilbert) and the *CardiffNLP*model also from Hugging Face (https://huggingface.co/cardiffnlp).

For each ML component, we specified the recommended information. For example, for the *DistilBERT*model, we defined the following elements: the model identifier (*e.g.*, distilbert/distilbert-base-uncased-finetuned-sst-2-english), version (*e.g.*, 1), model developer, model type (in this case, text classification), primary use (classification), application domains (*e.g.*, products, music, among others), performance metrics (*e.g.*, Accuracy), required computational resources (*e.g.*, 4-24GB of vRAM for GPU), microservice API endpoint (*e.g.*, http://localhost:5001), and interface type (REST API). Figure 3 shows an excerpt of the model card implementation for the ML-based Sentiment analysis component in VariaMos.

The information in the model card enables the selection of the most suitable machine learning model during product configuration, based on functional needs and the defined architecture, promoting effective reuse and traceability of models across different contexts.

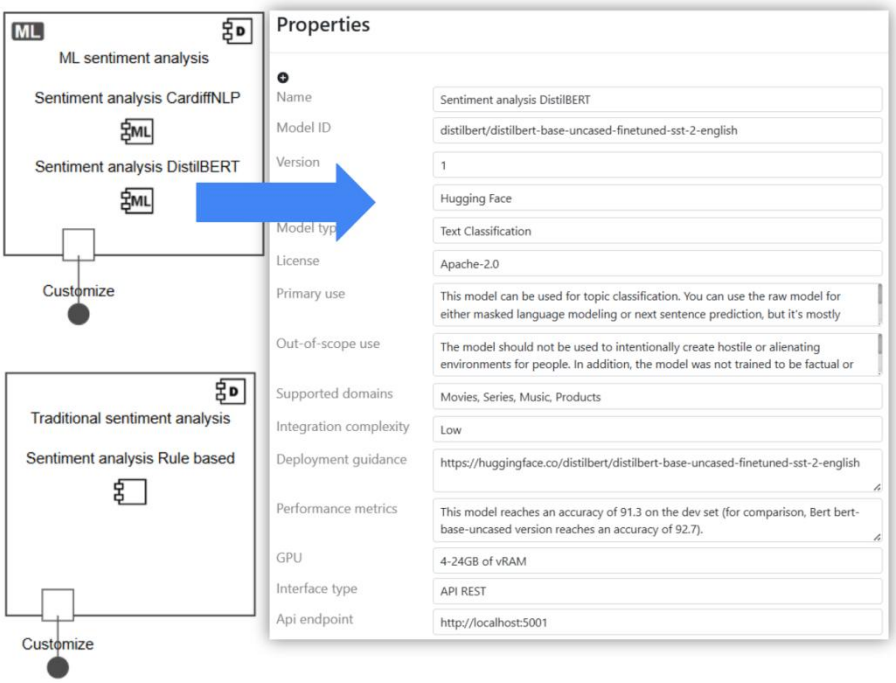

Figure 3. Excerpt from the model card for the sentiment analysis component, implemented in the VariaMos tool.

Recommendation 4: Implement Systematic ML Component Monitoring.

Given the inherently non-deterministic and data-dependent behavior of ML components, SPL engineers must design robust monitoring mechanisms capable of detecting performance degradation [21]. Operating at runtime, these mechanisms should continuously observe both model performance and business-critical signals, while being seamlessly integrated with drift detection and alerting processes to ensure resilient and self-adaptive system behavior.

Practical Implementation: To effectively implement this recommendation, SPL engineers should define a dedicated ML monitoring component for each domain component that incorporates ML capabilities. This component must specify the following attributes:

```
MLComponentMonitor: {
component_id: String,
monitoring_configuration: {
```

```
      metrics: Set[MonitoringMetric],
      frequency: TemporalSpecification,
data_collection_strategy: DataCollectionApproach,
baseline_establishment: BaselineDefinition
    },
threshold_definitions: {
performance_thresholds: Map[Metric, ThresholdSpec],
drift_detection_thresholds: Map[DriftType, ThresholdSpec],
business_impact_thresholds: Map[BusinessMetric, ThresholdSpec]
    },
intervention_strategies: {
alert_procedures: AlertSpecification
    }
}
```

The specification defines the structural requirements needed to establish consistent, interpretable, and actionable monitoring configurations. The key attributes of the monitoring specification are detailed below:

- **component_id**: Unique identifier of the monitored ML component. Used to record events, logs, and monitoring metrics.
- **monitoring_configuration**: Parameters that define what, how, and when monitoring is performed.
    - **metrics**: Set of key metrics for monitoring model performance. These metrics depend on the type of ML model (*e.g.*, classification [*F1 Score, AUC, Accuracy*], regression [*RMSE, MAE*], and recommendation [*Precision, Recall*]).
    - **frequency**: Frequency at which the model's status is evaluated. It may depend on the traffic rate or importance of the model (*e.g.*, *Hourly*: useful for high-volume production; *Daily*: balanced for general use; *EveryBatch*: suitable for batch systems; *RealTime*: when online processing is used).
    - **data_collection_strategy**: Method for collecting input data (for comparison and evaluation), predictions, and actual labels (if available) (*e.g.*, *StreamingLogs*: continuous online capture. (*e.g.*, *BatchLogs*: data collected in intervals; *ShadowDeployment*: evaluates without exposing to the user; *MiddlewareCapture*: collects from a proxy or wrapper).
    - **baseline_establishment**: Reference against which current metrics are compared. It can be a previous version or a historical average. (*e.g.*, *StaticThresholds*: defined by experts; *PrelaunchModelBaseline*: based on offline evaluation; *Rolling7DayAverage*: adaptive and dynamic).

- **threshold_definitions**: Set of thresholds that trigger alerts.
    - **performance_thresholds**: Thresholds over key model quality metrics.
    - **drift_detection_thresholds**: Statistical thresholds for detecting changes in the distribution (data drift, concept drift, prediction drift, etc).
    - **business_impact_thresholds**: Business metrics that may be impacted by the model, such as CTR, revenue, and churn.

- **intervention_strategies**: Defines actions to take if an anomaly or system degradation is detected.

- **alert_procedures**: Specification of the channel and form of alert to the responsible team (*e.g.*, SendMailToMLTeam, PushToPagerDuty).

Application in the E-commerce Example: In our running example, the sentiment analysis component can be continuously monitored to detect potential performance degradation, drift, or business impact issues.

In the case of the DistilBERT sentiment analysis ML component, the monitoring mechanism enables the definition of relevant performance metrics, such as Accuracy and Recall, as well as the establishment of an appropriate evaluation frequency (*e.g.*, daily assessments). It also allows the configuration of data collection strategies for evaluation purposes, such as streaming logs, and the definition of a reference baseline for comparison with current results, for instance, a seven-day moving average. Furthermore, the mechanism supports the specification of performance thresholds, including minimum and critical values for each metric, as well as the configuration of drift detection parameters that cover both data drift and concept drift. In addition, it facilitates the identification of business impact indicators, such as the number of misclassified negative reviews, and the definition of intervention strategies, including automated email alerts sent to the responsible team whenever an anomaly is detected. As illustrated in Figure 4, the configuration for the monitoring component of the Sentiment Analysis ML-component has been implemented in the VariaMos tool.

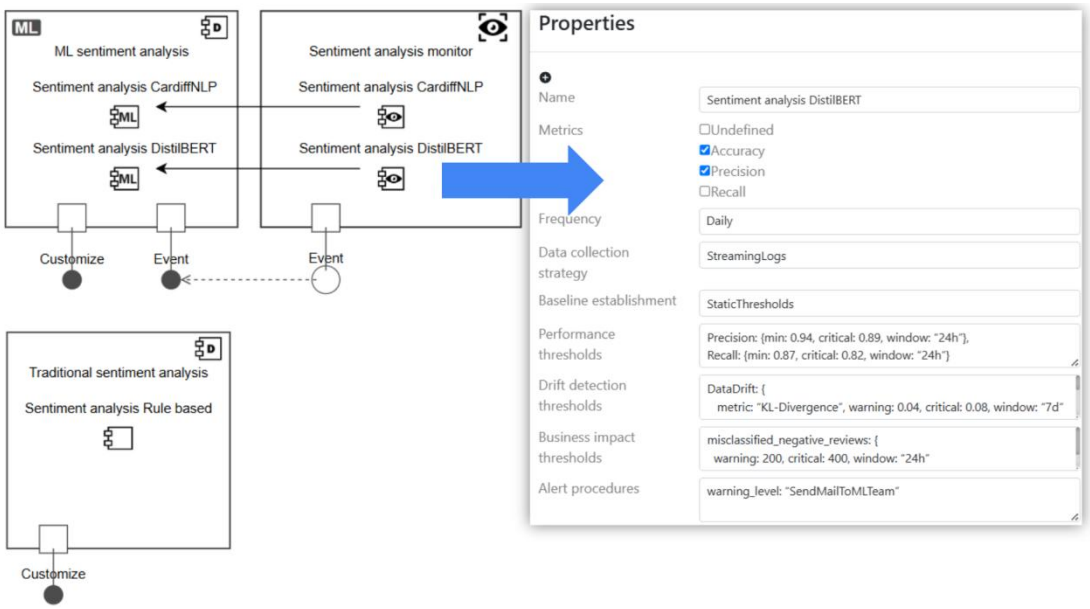

Figure 4. Component Monitoring configuration for Sentiment analysis ML-component, implemented in the VariaMos tool.

Recommendation 5: Implement Systematic ML Component Replacement Strategy.

During product configuration, an automated strategy should be established to update or replace ML components when performance degradation is detected. This requires the definition of an intervention mechanism that is triggered when the performance metrics of an ML component fall below predefined thresholds. The mechanism must support replacing the underperforming component with one of several alternatives: another ML model, a traditional software component, or, if appropriate, the temporary exclusion of the affected functionality from the system's execution flow.

Practical Implementation: To operationalize this recommendation, SPL engineers must define a replacement strategy component associated with each ML-enabled domain component. This component is responsible for responding to degradation alerts issued by the monitoring system and executing the actions defined in the replacement policy. The structure of the replacement strategy component can be formally specified as follows:

```
MLComponentReplacementStrategy = {
component_id: String,
replacement_hierarchy: {
primary_alternative: ComponentReference,
secondary_alternatives: List[ComponentReference],
fallback_strategy: FallbackApproach
  }
}
```

This specification defines the structure required to enable resilient and automated replacement mechanisms for ML components. The attributes are described below:

- **component_id**: Unique identifier of the ML component.

- **replacement_hierarchy**: Hierarchy of alternatives in case of model degradation.

    - **primary_alternative**: Component directly prepared to take over the current ML model.
    - **secondary_alternatives**: List of additional (less optimal) alternatives.
    - **fallback_strategy**: Emergency strategy to continue providing service with reduced capabilities (*e.g.*, AllowAll, ConservativeRuleBasedBlocking, RuleBasedBlocking, ManualReview, GracefulShutdown).

Application in the E-commerce Example: In an online retail SPL, a replacement strategy can be defined for the sentiment analysis component using both traditional and ML-based alternatives. To ensure system resilience, if no alternative component satisfies the required quality thresholds, a predefined fallback mechanism is activated, such as temporarily disabling the sentiment analysis feature within the process flow.

In our running example, the *DistilBERT*sentiment analysis component is associated with a monitoring component that tracks its performance. The replacement strategy allows defining a primary alternative, such as the *CardiffNLP*sentiment analysis component, and a secondary alternative represented by a traditional Rule-based component. A contingency strategy is also specified for cases in which none of the available alternatives meet the established quality criteria. In such situations, the system may temporarily block requests using a predefined rule set, ensuring controlled behavior in the presence of failures. The replacement strategy remains subscribed to the events generated by the monitoring system, enabling the automatic substitution of degraded components as soon as anomalies are detected.

In VariaMos, this mechanism ensures system resilience and operational continuity by automatically replacing underperforming components with viable alternatives, following a predefined hierarchy that responds to monitoring alerts and executes actions specified in the substitution policy. Figure 5 illustrates the replacement strategy implemented in VariaMos.

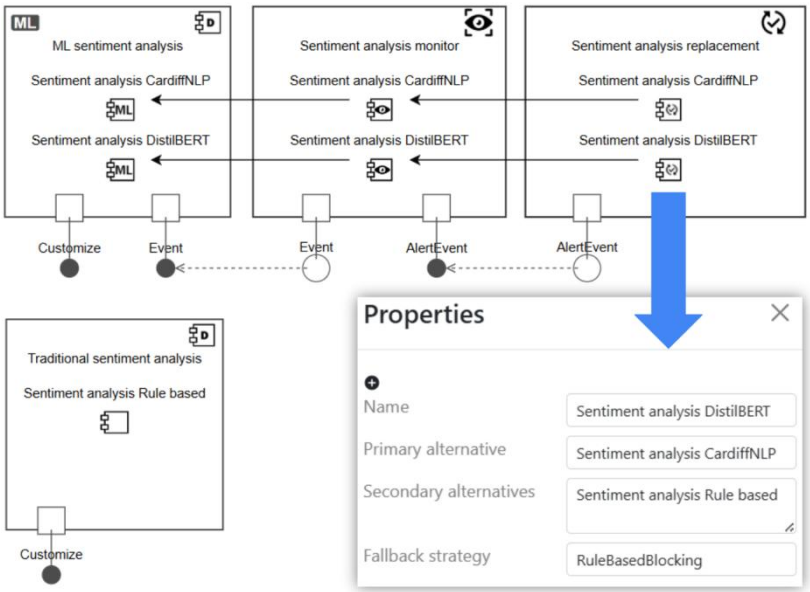

Figure 5. Configuration of the replacement strategy for the Sentiment analysis ML-component, implemented in the VariaMos tool.

Recommendation 6: Implement ML Component Orchestration.

Effective orchestration—the coordinated management and execution of ML components—in dynamic product configurations requires infrastructure that enables flexible model composition, state management between runs, and contextual integration. For this, we recommend:

- Use of modular ML pipelines, which allow integrating, monitoring, and scaling ML components in distributed environments.
- Intelligent orchestrators that dynamically adjust component activation according to contextual signals, business rules, or environmental conditions. Techniques such as context-aware scheduling can be applied.
- Functional decoupling of components, promoting a microservices-based architecture to facilitate model replacement, enhancement, or re-trainability without altering the overall configuration.
- Instrumentation for traceability and versioning: employ systems that record training data, parameters, results, and decisions made by each component to facilitate audits and optimization.

Practical Implementation: To operationalize this recommendation, SPL engineers must define a dedicated orchestration layer that governs the lifecycle, dependencies, and interactions of ML components. This orchestration must support declarative workflows, dynamic adaptation policies, and seamless integration with monitoring systems. The following schema defines a formal representation of such an orchestration-aware product configuration:

```
ProductConfiguration = {
configuration_id: String,
feature_binding: Map[Feature, ComponentBinding],
workflow_specification: {
component_graph: DirectedAcyclicGraph[Component, DataFlow],
execution_constraints: Set[Constraint],
quality_objectives: Map[QualityAttribute, Objective],
```

```
resource_allocations: Map[Component, ResourceAllocation]
  }, adaptation_policies: {
monitoring_configuration: MonitoringPolicy,
replacement_triggers: Set[ReplacementTrigger],
quality_negotiation: QualityNegotiationStrategy,
performance_optimization: OptimizationPolicy
  }, validation_requirements: {
functional_tests: Set[TestSpecification],
performance_benchmarks: Set[BenchmarkTest],
quality_assertions: Set[QualityAssertion],
compliance_checks: Set[ComplianceCheck]
  }
}
```

This schema defines the structural and behavioral dimensions of a configurable product instance. Its modular design supports precise, verifiable, and adaptive configuration management across a wide range of variability. The key components are described below:

- **configuration_id**: A unique identifier assigned to the product configuration instance.

- **feature_binding**: A mapping between product features and their corresponding component implementations. This allows resolution of variability by specifying which components realize which features in a given configuration.

- **workflow_specification**: Captures the operational logic of the product.

    - **component_graph**: A directed acyclic graph (DAG) that defines the data flow and execution dependencies among software and ML components.
    - **execution_constraints**: A set of logical or resource-based constraints that govern component execution (*e.g.*, timing, sequencing).
    - **quality_objectives**: Specifies target values for quality attributes, such as accuracy, latency, and energy consumption.
    - **resource_allocations**: Assigns computational resources (*e.g.*, CPU, memory, GPU) to each component to ensure operational feasibility.

- **adaptation_policies**: Define the runtime behavior of the product under varying operational conditions:

    - **monitoring_configuration**: Indicates how system performance is monitored during execution.
    - **replacement_triggers**: Defines conditions under which components should be replaced.
    - **quality_negotiation**: Specifies strategies for balancing competing quality attributes under constraints.
    - **performance_optimization**: Policies for dynamically optimizing performance based on monitored feedback.

- **validation_requirements**: Ensures that configured products meet their intended goals and regulatory requirements:

    - **functional_tests**: Set of specifications for functional correctness.
    - **performance_benchmarks**: Benchmark tests that measure system performance under predefined workloads.

- **quality_assertions**: Verifiable and testable statements specifying the quality attributes that a configured product is required to meet.
- **compliance_checks**: Formal |checks to ensure adherence to standards, certifications, or domain-specific regulations.

Application in the E-commerce Example: In an online retail SPL, a dynamic product configuration may include ML components for personalized recommendations, fraud detection, and sentiment analysis. The component bindings for each product instance can vary significantly based on factors such as the target audience, expected transaction volume, and specific regional compliance mandates.

4.4. Dynamic Product Configuration

Incorporating ML components during product configuration adds depth to variability and intelligence of SPL. However, configuration decisions must balance multiple competing objectives, such as performance, cost, and reliability, often under shifting operational conditions.
Recommendation 7: Establish Multi-Objective Configuration Optimization.

To enhance the adaptability and performance of ML-enabled SPLs, it is essential to establish multi-objective configuration optimization mechanisms. This approach enables organizations to simultaneously evaluate and balance competing concerns, including accuracy, latency, resource consumption, interpretability, and ethical constraints. By leveraging advanced optimization techniques such as Pareto efficiency or evolutionary algorithms, teams can generate configuration sets that meet diverse stakeholder requirements without compromising system integrity. Implementing multi-objective optimization also promotes continuous improvement, enabling dynamic reconfiguration as environments evolve or model behaviors drift over time.

Practical Implementation: To operationalize multi-objective configuration optimization, it is first necessary to formalize a set of competing objectives, such as model accuracy, latency, resource utilization, interpretability, and compliance with ethical standards, into quantifiable metrics. The configuration space should encompass both system-level parameters and ML-specific settings, including hyperparameters and pipeline structures. Exploring trade-offs across this space can be conducted using optimization techniques such as evolutionary algorithms (*e.g.*, NSGA-II), Bayesian multi-objective methods, or Pareto-based analysis. Configurations are evaluated through simulation or benchmarking, producing Pareto-optimal sets that offer balanced solutions. These sets can be visualized or presented through decision-support interfaces to facilitate selection based on dynamic stakeholder priorities and preferences. Finally, integrating optimization processes within CI/CD pipelines ensures continuous reconfiguration in response to model drift or changing operational constraints.

4.5. Product Derivation and Validation

This phase considers the methodology for deriving specific products from a configurable architecture, detailing how optimization criteria and stakeholder requirements guide the selection process. It also describes the validation mechanisms employed to ensure that the resulting products meet expected standards of functionality, performance, and reliability prior to deployment.

Recommendation 8: Implement validation and testing strategies specifically designed for ML-enhanced products.

Validation and testing strategies should incorporate both functional and non-functional assessments, including unit and integration testing, model performance evaluation across diverse datasets, fairness audits, and resource utilization benchmarking. In addition, these strategies should extend to include ML-specific validation approaches, such as statistical performance validation, bias detection testing, adversarial robustness assessment, and long-term stability verification. It must also support automated validation pipelines integrated into CI/CD workflows, enabling continuous monitoring and the rapid detection of anomalies, drift, or compliance violations.

Practical Implementation: To implement this recommendation, the first step is to configure the derivation. This involves selecting binary features and determining the quality distributions for ML components. It is essential to establish optimization criteria and stakeholder requirements to guide the automated selection of ML components. This ensures that each product is customized to meet its use case requirements. Once a product is derived, the unit tests, integration tests, and non-functional requirement tests must be executed. The process is further enhanced with ML-specific validations such as bias detection, adversarial robustness assessments, and long-term stability verification to address the unique vulnerabilities of machine learning models. The overall goal is to ensure that the derived products meet all expected standards of functionality, performance, and reliability prior to deployment. Finally, to ensure long-term reliability, all of these validation strategies are integrated into automated CI/CD pipelines.

In conclusion, the proposed framework's recommendations, which encompass the entire lifecycle of ML-enhanced Software Product Lines (SPLs), have been demonstrated through the running example and their implementation within the VariaMos web tool. The running example illustrates the framework's feasibility and applicability, showing how it supports the development of adaptive and intelligent product lines. Furthermore, these recommendations are currently being applied to the development of a proof-of-concept SPL for a text editor, enabling the empirical validation of the framework's practical effectiveness.

5. RELATED WORK

The intersection of SPL engineering and ML represents an emerging research area that builds upon established foundations in both domains. Traditional SPL engineering, formalized through seminal work by Clements, Mazo, and Pohl, respectively [8], [17], [18], has established comprehensive methodologies for systematic software reuse through domain engineering and application engineering processes. The Feature-Oriented Domain Analysis (FODA) approach introduced by Kang [9] and subsequent advances in variability management [8], [22]provide robust frameworks for managing product family complexity. However, these approaches fundamentally assume component determinism and behavioral predictability, creating significant gaps when dealing with probabilistic ML components.

Parallel developments in ML engineering have addressed the unique challenges of ML-enabled systems through comprehensive frameworks for technical debt management [21], engineering practices [23], and quality assurance approaches [24]. The emergence of systematic documentation practices, as exemplified by Model Cards [19] and behavioral testing frameworks [25], represents substantial progress in ML system engineering. Recent systematic reviews by Martínez-Fernández [4] and empirical studies by Nahar and Ribeiro, respectively [5], [25] have documented collaboration challenges and the complexity of requirements engineering specific to

ML systems. Nevertheless, this body of work predominantly focuses on standalone ML systems or monolithic application contexts, with limited consideration of systematic reuse frameworks.

Architectural approaches for ML integration have evolved toward microservices-based patterns [26] and adaptive system frameworks [13], [27], while dynamic SPL research [8], [28], [29]has explored evolution and adaptation in product line contexts. However, existing approaches have not systematically addressed the unique requirements of ML components within SPL environments, including cross-product consistency management, shared component instance coordination, and the specific adaptation patterns required for probabilistic components subject to performance degradation and concept drift.

Current literature reveals critical limitations when applied to ML-enhanced SPL contexts. Traditional SPL methodologies assume behavioral predictability, which is incompatible with the probabilistic nature of ML components. In contrast, ML engineering approaches lack systematic frameworks for ensuring cross-product consistency and shared component management. Existing documentation frameworks do not provide mechanisms for reusability assessment required for SPL component selection, and current adaptive system approaches do not address ML-specific degradation patterns and monitoring requirements.

This work addresses these fundamental gaps by providing the first comprehensive framework specifically designed to integrate ML components within SPLs, while preserving the benefits of systematic reuse. Unlike existing approaches that treat ML components as standalone services or apply ad-hoc integration patterns, our framework systematically extends established SPL methodologies with ML-specific concepts, including probabilistic feature modeling, degradation-aware component characterization, adaptive architectural patterns, and dynamic configuration optimization. The framework proposed in this paper provides concrete specifications, including formal orchestration languages (MCOSL), systematic monitoring frameworks, and multi-objective optimization approaches, enabling practitioners to maintain an engineering discipline and leverage systematic reuse advantages while effectively utilizing ML capabilities across products derived from product lines.

6. CONCLUSIONS AND FUTURE WORK

The integration of ML components into SPLs presents new challenges that traditional modeling techniques are not equipped to address. By addressing the variability and uncertainty inherent in ML components, this approach lays the groundwork for bridging the gap between SPLE and AI-based software development.

In this paper, we propose a framework that supports the inclusion of ML components in SPLs, facilitating systematic reuse, customization, and evolution. Our contribution consists of a **specification-oriented approach** that guides the integration of ML-based functionalities into SPLs, along with a set of recommendations and practical implementations. The framework extends existing variability management approaches to support **ML-aware configuration and reuse**. The framework is structured around five interconnected phases that encompass the entire lifecycle of ML-enhanced SPLs: ML-aware domain analysis, Adaptive architecture design, ML-aware domain implementation, Dynamic product configuration, and Product derivation and validation of its resulting products. The framework's feasibility is demonstrated through an implementation in the VariaMos tool and a case study that validates its applicability to real-world scenarios.Initial empirical findings, obtained by applying these recommendations to two distinct SPLs—an e-commerce SPL and a text editor SPL—suggest that this comprehensive documentation approach facilitates informed decision-making across the entire ML component lifecycle. This process spans from initial model selection to deployment and ongoing monitoring.

Furthermore, Model Cards support regulatory compliance and risk management by providing auditable documentation of model characteristics and decision rationale, which contributes to the development of more accountable and trustworthy ML systems. Although these preliminary results are promising, further experimentation and implementation improvements are needed to fully assess the actual value and impact of this proposal in production environments. Future work involves evaluating the proposed strategy in real-world industrial domains, including a detailed cost-benefit analysis, extending the capabilities of the VariaMos tool, and exploring its applicability to AI components beyond ML.

ACKNOWLEDGMENTS

This research was supported by the University of Antioquia, Colombia, through the Committee for the development of research – CODI (PRV2022-52951), and the ENSTA, France.

REFERENCES

[1] N. Ahmed and N. Shakoor, "Advancing agriculture through IoT, Big Data, and AI: A review of smart technologies enabling sustainability," *Smart Agricultural Technology*, vol. 10, no. 100848, Mar. 2025, doi: 10.1016/j.atech.2025.100848.

[2] G. Anthes, "Artificial intelligence poised to ride a new wave," *Communications of the ACM*, vol. 60, no. 7, pp. 19–21, June 2017, doi: 10.1145/3088342.

[3] G. Giray, "A software engineering perspective on engineering machine learning systems: State of the art and challenges," *Journal of Systems and Software*, vol. 180, no. 111031, Oct. 2021, doi: 10.1016/j.jss.2021.111031.

[4] S. Martínez-Fernández *et al.*, "Software Engineering for AI-Based Systems: A Survey," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–59, Apr. 2022, doi: 10.1145/3487043.

[5] N. Nahar, S. Zhou, G. Lewis, and C. Kästner, "Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, May 2022, pp. 413–425. doi: 10.1145/3510003.3510209.

[6] N. Nahar, H. Zhang, G. Lewis, S. Zhou, and C. Kästner, "A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners," in *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, May 2023, pp. 171–183. doi: 10.1109/CAIN58948.2023.00034.

[7] L. Cobaleda, J. Carvajal, P. Vallejo, A. López, and R. Mazo, "Enhancing Software Product Lines With Machine Learning Components" in *Computer Science & Information Technology (CS & IT)*, vol. 15, no. 20, pp. 73-94, Oct. 2025,doi: 10.5121/csit.2025.152006

[8] R. Mazo, Ed.,*Guía para la adopción industrial de líneas de productos de software*. Universidad Eafit, 2018.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Carnegie-Mellon University, 1990.

[10] L. Dounas, R. Mazo, C. Salinesi, and O. El Beqqali, "Continuous monitoring of adaptive e-learning systems requirements," in *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*, Nov. 2015, pp. 1–8. doi: 10.1109/AICCSA.2015.7507210.

[11] A. Achtaich, N. Souissi, C. Salinesi, R. Mazo, and O. Roudies, "A Constraint-based Approach to Deal with Self-Adaptation: The Case of Smart Irrigation Systems," *IJACSA*, vol. 10, no. 7, 2019, doi: 10.14569/IJACSA.2019.0100727.

[12] A. Achtaich, N. Souissi, R. Mazo, O. Roudies, and C. Salinesi, "A DSPL Design Framework for SASs: A Smart Building Example," *EAI Endorsed Transactions on Smart Cities*, vol. 3, no. 8, June 2018.

[13] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," *Journal of Systems and Software*, vol. 91, pp. 24–47, May 2014, doi: 10.1016/j.jss.2013.06.034.

[14] C. Dumitrescu, R. Mazo, C. Salinesi, and A. Dauron, "Bridging the gap between product lines and systems engineering: an experience in variability management for automotive model based systems engineering," in *Proceedings of the 17th International Software Product Line Conference*, in SPLC '13. New York, NY, USA: Association for Computing Machinery, Aug. 2013, pp. 254–263. doi: 10.1145/2491627.2491655.

[15] R. Mazo, S. Assar, C. Salinesi, and N. Ben Hassen, "Using software product line to improve ERP engineering : literature review and analysis," *Latin-American Journal of Computing*, vol. 1, no. 1, p. ., Oct. 2014.

[16] V. Indykov, "Component-based Approach to Software Engineering of Machine Learning-enabled Systems," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, in CAIN '24. New York, NY, USA: Association for Computing Machinery, June 2024, pp. 250–252. doi: 10.1145/3644815.3644976.

[17] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley, 2001.

[18] K. Pohl, G. Böckle, and F. Van Der Linden, *Software Product Line Engineering*. Berlin, Heidelberg: Springer, 2005. doi: 10.1007/3-540-28901-1.

[19] M. Mitchell *et al.*, "Model Cards for Model Reporting," in *Proceedings of the Conference on Fairness, Accountability, and Transparency*, in FAT* '19. New York, NY, USA: Association for Computing Machinery, Jan. 2019, pp. 220–229. doi: 10.1145/3287560.3287596.

[20] T. R. Toma, B. Grewal, and C.-P. Bezemer, "Answering User Questions About Machine Learning Models Through Standardized Model Cards," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1488–1500. doi: 10.1109/ICSE55347.2025.00066.

[21] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, "Hidden Technical Debt in Machine Learning Systems," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2015.

[22] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th international conference on Software engineering*, in ICSE '08. New York, NY, USA: Association for Computing Machinery, May 2008, pp. 311–320. doi: 10.1145/1368088.1368131.

[23] S. Amershi*et al.*, "Software Engineering for Machine Learning: A Case Study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Montreal, QC, Canada: IEEE, May 2019, pp. 291–300. doi: 10.1109/ICSE-SEIP.2019.00042.

[24] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec. 2017, pp. 1123–1132. doi: 10.1109/BigData.2017.8258038.

[25] M. T. Ribeiro, T. Wu, C. Guestrin, and S. Singh, "Beyond Accuracy: Behavioral Testing of NLP Models with CheckList," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, July 2020, pp. 4902–4912. doi: 10.18653/v1/2020.acl-main.442.

[26] S. Newman, Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2021.

[27] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004, doi: 10.1109/MC.2004.175.

[28] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic Software Product Lines," *Computer*, vol. 41, no. 4, pp. 93–95, Apr. 2008, doi: 10.1109/MC.2008.123.

[29] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, "Constraint Programming as a Means to Manage Configurations in Self-Adaptive Systems," *Special Issue in IEEE Computer Dynamic Software Product Lines*, pp. 1–12, Dec. 2012.

AUTHORS

Luz-Viviana Cobaleda is an Associate Professor at the University of Antioquia (Colombia). She holds a Ph.D., M.Sc., and B.Sc. in Engineering from the same university, and a Specialization in Software Engineering from EAFIT University. Her research focuses on software engineering, with publications in international venues and participation in collaborative projects.

Andrés López is a systems engineer. He graduated in 2013 with a degree in Systems Engineering from the University of Antioquia (Colombia) and in 2018 obtained a Master's in Engineering from EAFIT University (Colombia). He is currently pursuing a joint Ph.D. in Sciences pour l'ingénieur et le numérique à l' École Nationale Supérieure de Techniques Avancées – ENSTA (France) and in Electronic and Computer Engineering at the University of Antioquia.

Paola Vallejo is a Systems Engineer who graduated from Universidad EAFIT in 2012. She got her Master's degree (Human Computer Centered Systems) at École Nationaled'Ingénieurs de Brest - France in 2012. She received the Ph.D. degree in Computer Science from Université de Bretagne Occidentale, France, in 2015. She is currently a full professor at Universidad EAFIT.

Raúl Mazo is a Franco-Colombian engineer who received his Engineering degree in Informatics from the University of Antioquia (Colombia) in 2005, and later earned an M.S. in Information Systems, a Ph.D. in Computer Science, and the Habilitation à Diriger des Recherches (HDR) from the University Panthéon-Sorbonne (France) in 2008, 2011, and 2018, respectively. He is currently a Full Professor at the École Nationale Supérieure de Techniques Avancées (ENSTA).

Julian Carvajal is a Colombian software engineer in training and a Systems Engineering student at the University of Antioquia (Colombia). He has professional experience as a software developer, with a particular focus on building educational video games for preschool children and contributing to research-driven software projects.