Feedback analysis in software product line forked developments

Anonymous Author(s)

Abstract

Software Product Lines (SPLs) enable the reuse of software components or assets to generate a family of related products. Sometimes, an SPL evolves into parallel developments (forks) to meet new requirements. However, these forks do not always stay synchronized with the central development, e.g., features can be added, removed, or changed in the forked projects. In DevOps practices, feedback analysis plays a central role in improving both software quality and delivery processes. DevOps feedback analysis evaluates data from the delivery pipeline and users to improve the software and its deployment process continuously. Despite its importance, feedback analysis has been underexplored in the context of SPLs. In this paper, we propose an approach to automate feedback analysis in forked software product line developments that can allow us to assist decision-making processes in answering questions such as: Which features need more testing? What new features can be incorporated? Which ones require refactoring? Which ones cause more issues in production? Information can be gathered from various data sources such as source code repositories, bug tracking systems, or continuous integration pipelines. To the best of our knowledge, this is the first proposal using information from forked SPL developments for feedback analysis. To evaluate the proposal, we automatically analyzed 25 forks of an open-source SPL that helped us assist in decision-making tasks, showing the feasibility of our proposal.

CCS Concepts

• **Software and its engineering → Software evolution**.

Keywords

SPL, fork analysis, feature evolution, traceability, feedback automation

ACM Reference Format:

Anonymous Author(s). 2025. Feedback analysis in software product line forked developments. In *Proceedings of 29th International Systems and Software Product Line Conference (SPLC'25)*. ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software Product Lines (SPLs) provide a structured approach to reuse software assets, enabling the systematic derivation of related product variants. This paradigm aims to reduce both development effort and time-to-market while enhancing product consistency and quality. SPLs support scalability and adaptability to different markets by allowing for the customization of final products based on asset reuse.

An SPL may sometimes develop into parallel developments or *forks* [5]. These forks often arise in response to new requirements that are incompatible with the existing architecture or variability constraints. The reasons for these forks can vary, including customer-specific constraints, new technologies' introduction, or market shifts [13]. While forks enable quick and localized adaptations, they can also lead to independent evolution, which may fragment the main development line.

This divergence hinders maintainability, limits asset reuse, and introduces architectural inconsistencies [5]. Changes made in these parallel branches are often not reintegrated into the central repository, encouraging the emergence of ad hoc solutions and leading to a loss of control over the product's common evolution. Centralized variability management allows us to capture these new requirements within the formal model. This approach prevents the uncontrolled proliferation of versions [29] and ensures the traceability and consistency of the SPL. Whenever possible, merging forks into the mainline is essential for maintaining the sustainability of the software ecosystem [34].

Continuous feedback analysis is a cornerstone of DevOps, facilitating early error detection, quality improvement, and faster deployment cycles. However, automating this process becomes more complex in fork-based development environments like those in SPLs. In such scenarios, feedback becomes fragmented across repositories and teams, posing challenges for systematic collection and analysis. In addition, the identification and traceability of specific features [43] within forks require specialized approaches to manage variability and maintain consistency in product development [9].

Although feedback loops are essential in DevOps [17], their role in SPLs is underexplored. SPL research has focused on variability management, configuration, and evolution [32, 38]. Few works address systematically collecting or analyzing user or developer feedback across product variants. Existing SPL tooling lacks mechanisms for continuous feedback integration. Fork-based SPL development further fragments feedback, increasing the challenge [43]. This gap limits data-driven decision-making in SPL engineering.

When parallel developments are not aligned with the main line of progress, tracking the behavior and evolution of features becomes difficult. This lack of synchronization leads to uncertainty about which parts of the system should be developed further, maintained, or removed. In the absence of traceability, teams lack reliable information for effective maintenance, test planning, and change impact analysis. Consequently, the variability model no longer accurately reflects the operational state of the SPL and becomes disconnected from the actual state of the code [4].

We define *automatic feedback collection* as the extraction of development data—such as commits, issues, and pull requests—from

SPLC'25, A Coruña, Spain

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

forked repositories, with the aim of supporting the controlled evolution of the original SPL. Analyzing this feedback informs decision-making at the feature level within the original SPL. Resulting actions—such as feature addition, refactoring, decomposition, or removal—are grounded in evidence derived from development, testing, and integration activities. This approach allows the SPL to evolve coherently, integrating knowledge generated in derivative or parallel developments, thus avoiding loss of synchronization or proliferation of isolated solutions. We formulate the following research questions to guide our contribution:

- *RQ1. How can development activity from derivative forks be collected and mapped to features?*
  This question explores the design of a data model that captures the relationship between features, data sources, and tags to enable decision-making.
- *RQ2. How can the analysis of traceability data support informed decision-making in testing, feature evolution, and CI/CD pipelines within an SPL?*
  This question explores how structured feedback extracted from forks can help identify testing gaps, guide feature-level changes, and improve integration workflows.

By answering these questions, we aim to provide a systematic approach to SPL evolution that leverages evidence from distributed development activities, thus bridging the gap between isolated forks and the central variability model.

The remainder of this paper is structured as follows: Section 2 identifies the types of feedback in software engineering, the software mining repositories, the evolution of SPLs and DevOps; Section 3 details our proposal for automatic feedback collection through a systematic flow; Section 4 performs an evaluation of the proposal, first with a proof of concept on one repository and then extending it to all forks; Section 5 details some threats to the validity of the proposal, highlighting weaknesses; Section 6 discusses kinds of forks and their limitations; Section 7 shows related work on automatic feedback collection; and Section 8 presents conclusions and future work.

2 Background

This work explores three key areas: feedback in development processes, mining software repositories, and the evolution of software product lines. It defines and classifies different types of feedback and explains their role in agile methodologies. It also outlines techniques to extract information from historical data in repositories. Finally, it examines how to manage evolution and maintain traceability in product lines.

2.1 Feedback analysis and DevOps practices

In software engineering, feedback refers to information obtained from both software execution and stakeholder interactions, which serves to guide the improvement of the product and the development process. Feedback helps improve the product or the development process. For example, end-user feedback on deployed applications can be leveraged to refine requirements and identify latent defects. This concept is key in Agile and DevOps [40]. These methodologies focus on fast, continuous feedback cycles. They allow developers to adjust quickly to real needs [39].

Academic literature has classified feedback into several categories [19, 27]. In software engineering, feedback can be explicit (provided consciously by users), implicit (derived from use without direct intervention), or continuous (permanently integrated into the development process), all of which serve to validate and guide the evolution of a software product.

- *Explicit feedback*. Explicit feedback refers to the feedback that users are consciously providing. This feedback can include activities such as filling out surveys, submitting bug reports, or participating in interviews with developers. This type of feedback gathers users' subjective opinions and is typically collected through direct interactions, such as interviews, questionnaires, or suggestions in forums [27].
- *Implicit feedback*. Implicit feedback is collected indirectly by inferring it from user behavior, often without the user's awareness. For instance, data from application usage, event logs (telemetry), or automatically recorded user clicks are all examples of implicit feedback. This type of feedback reflects objective user behavior, highlighting how and when users utilize specific features. [27].
- *Continuous feedback*. Continuous feedback focuses less on user intentionality and more on the regularity and effectiveness with which feedback is integrated into iterative development cycles. Agile methodologies and continuous delivery practices prioritize incorporating feedback at all stages [20], avoiding the tendency to wait until the end of the project. This approach necessitates continuous monitoring of the system in production, which includes collecting metrics and user feedback during each iteration. The development team utilizes this information to make prompt decisions [36].

Modern development practices such as DevOps operationalize continuous feedback through automation and integration. Continuous Integration (CI) enables teams to merge code changes frequently into a shared repository, triggering automated builds and tests that detect issues early [24]. Continuous Deployment (CD) extends this by automating the release of validated changes to production environments. These practices reduce manual overhead, accelerate deployment timelines, and enhance the responsiveness of feedback mechanisms [10, 35].

DevOps emphasizes short iteration cycles and relies on monitoring tools to collect runtime data, user behavior, and system metrics. This infrastructure supports the continuous collection and analysis of feedback from production environments, enabling teams to react quickly to failures, usage patterns, or changing requirements. Contemporary research on DevOps highlights its shift toward greater cross-functional collaboration and automation, incorporating AI-assisted techniques for tasks such as automated testing and deployment oversight [16].

2.2 Mining software repositories

Software repository mining is a field in software engineering. It analyzes historical data from development repositories. Its objective is to extract actionable insights regarding software evolution, quality, and development practices [23].

These repositories, such as GitHub, contain valuable traces of developer activity, including commit histories, issue discussions, pull requests, and test results [33]. Analyzing this data reveals patterns in software evolution, team collaboration dynamics, and potential fault-prone areas or improvement opportunities [42]. Several techniques and analysis goals have emerged in this field:

- *Analysis of commit history*. It identifies patterns of code evolution, locates errors (e.g., using the SZZ algorithm [23]), and detects coupled changes that reveal non-explicit logical dependencies [23].
- *Analysis of issues and defects*. Studies bug reports and enhancement requests to predict faulty modules, measure process metrics, and prioritize maintenance [11].
- *Analysis of pull requests and revisions*. Examines change proposals and their review cycle to understand collaborative dynamics and their impact on software quality [25].
- *Analysis of developer contributions and profiles*. It quantifies individual contributions and uncovers the project's sociotechnical structure, including identification of key contributors and risk associated with knowledge concentration (e.g., the bus factor) [2].
- *Analysis of code evolution and architecture*. Reconstructs the structural evolution of the software, identifies hot spots, and analyzes the co-evolution of components [23].

2.3 Software product line evolution

SPL evolution involves controlled changes to reusable assets. The goal is to adapt to new requirements, technologies, or markets. A key research focus is the evolution of feature models. Researchers have developed analysis techniques to detect inconsistencies after changes, such as dead features. They have also proposed catalogs of safe refactoring for feature models. These ensure that the set of valid products remains the same. [37].

SPL evolution impacts a wide range of artifacts, including source code, configuration files, documentation, and testing infrastructure. It is important to keep feature models and implementations in sync. This sync avoids inconsistencies and divergence [26]. Some approaches address this challenge by establishing explicit traceability links between features and implementation artifacts, making their relationships visible and maintainable throughout the lifecycle. Others rely on static code analysis to automatically infer these links by examining code dependencies, structural patterns, or historical changes [21]. Other works have addressed the parallel evolution of variants and the platform [22]. These works propose mechanisms to propagate changes between them without breaking compatibility. Although secure evolution patterns and supporting tools have been proposed, their adoption in industrial contexts remains limited due to complexity, scalability issues, and integration challenges [6]. These industrial case studies in embedded systems and enterprise software have shown the practical challenges: increasing model complexity, erosion of variability, or misalignment between requirements and architecture [3]. Despite these advances, the literature highlights some gaps. There is a need for more large-scale empirical studies. Better integration between modeling tools and real development practices is also needed [7]. Perdek and Vranic introduce a fully automated, minimalist approach to SPL evolution focused on code fragments [31]. Their method manages variability autonomously. It uses annotations and configurable strategies. The approach generates multiple views—code, graphs, logs, images, and ontologies. These views support simulation, analysis, and traceability.

3 Automatic feedback collection

Figure 1 outlines a process in which domain engineers initially develop the SPL by defining core assets and maintaining the variability model. Over time, application engineers create forks of the SPL to implement specific requirements or address limitations not covered by the original version.

These forks develop independently and may involve multiple developers working simultaneously. Each fork produces development artifacts—such as commits and issues—that can be systematically collected and analyzed. We identify three primary sources from which technical feedback can be obtained.

- *Version control systems (VCS).* The repository metadata stores the complete history of commits, file modifications, and authorship, allowing us to track how the fork evolves over time [41].
- *Project tracking software.* Platforms that support collaboration workflows—such as pull or merge requests, issue tracking, and review discussions—provide insights into development coordination, integration efforts, and review dynamics [8].
- *CI/CD infrastructure.* Continuous integration and deployment systems generate logs from automated builds, tests, and deployments. These artifacts expose which features are deployed, which fail, and under which conditions or configurations [14].

The collected data is used to generate what we refer to as a *delta version*, representing the cumulative evolution within a fork over a specific period (typically 4 to 6 months) [44] and includes changes such as added, removed, or modified features.

In a general setting, it is first necessary to identify and localize features within the codebase to track feature evolution. This identification often requires an intermediate abstraction step to infer these feature boundaries from code changes using static analysis, dependency graphs, or naming conventions. In our case study, however, the set of features is already explicitly defined in a structured format, allowing us to associate development activity with known features directly. While this structure facilitates direct mapping, less structured systems may require additional feature identification techniques such as static analysis or pattern recognition.

All delta versions are consolidated into a centralized structure, referred to as the *feedback log*. This log serves as the foundation for consolidating technical evidence across forks, enabling feature-centric analysis to inform the evolution of the SPL. By leveraging this log, maintainers can systematically identify relevant modifications in derivative developments and guide the adaptation of the central product line in a traceable and coherent way.

Figure 2 presents an overview of the automatic feedback collection process designed to support SPL evolution from forked developments. This comprehensive process involves the following stages:

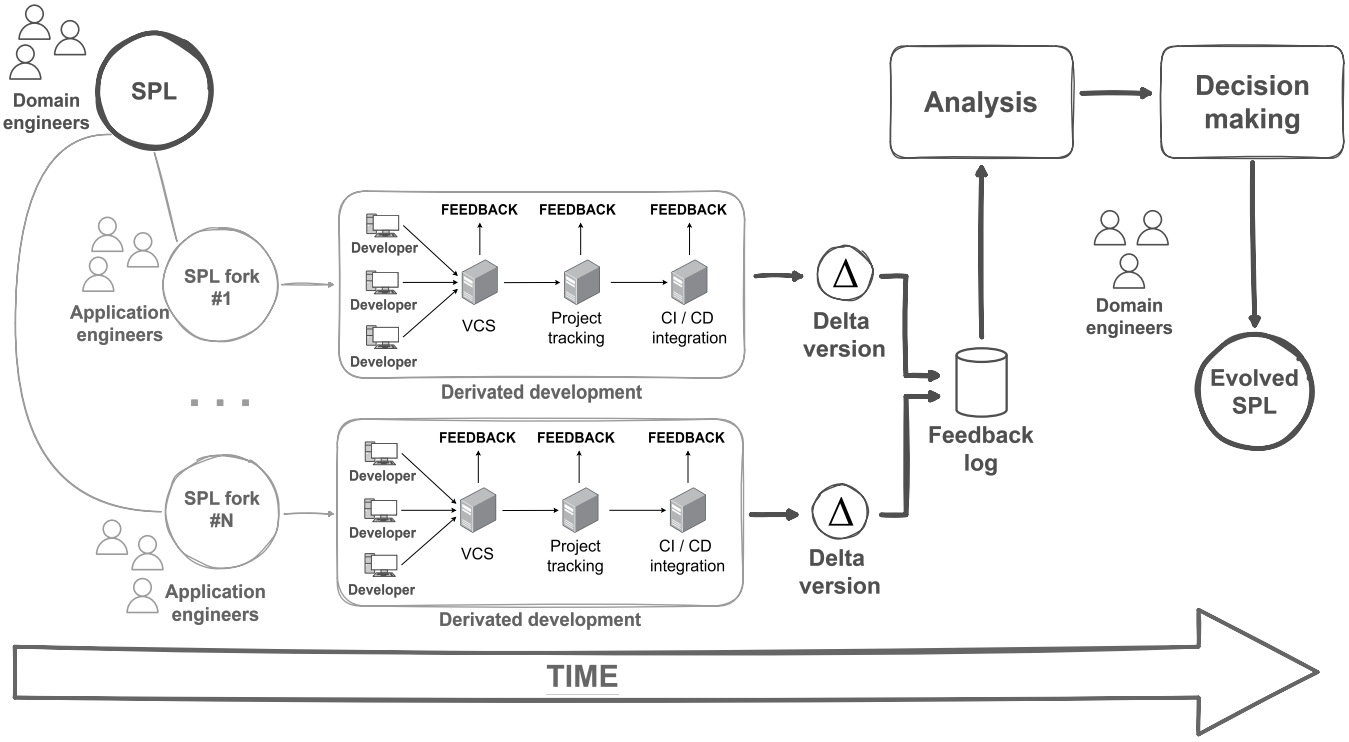

Figure 1: Evolution of an SPL based on forked developments

(1) *Initial versions of forks*: The process starts with one or more forks created from the original SPL, representing parallel developments. Each fork begins from an initial version of the SPL, adapting it to specific needs or requirements.
(2) *Data source*: Feedback from each fork is automatically gathered through development activities, such as commits, issues, and pull requests. These provide granular details of the development progress and changes made.
(3) *Data processing*: Collected data from each fork is systematically filtered, processed, and consolidated to generate a coherent representation of changes. This stage produces a *Delta version*, which encapsulates all relevant modifications, including new, extended, or refactored features.
(4) *Feedback log*: All delta versions data source from multiple forks are collected into a centralized *Feedback log*, forming a comprehensive repository of changes and enhancements derived from various forks. This log serves as a unified data structure aggregating feedback from multiple forks, enabling feature-centric analysis.
(5) *Analysis*: Data from the feedback log is analyzed by categorizing it into testing-related questions, feature-related questions, and CI/CD pipeline questions. This analysis identifies areas of improvement, recurrent issues, and feature requirements.
(6) *Decision-making*: Based on analytical outcomes, informed decisions are taken to evolve the SPL. Possible actions include testing features, adding, extending, refactoring, splitting, or deleting features, and modifying CI/CD pipelines.

3.1 Initial versions of forks

SPL evolution often initiates through the creation of forks, which represent independent development trajectories derived from the original product line. Forks represent parallel developments that address requirements or enhancements not included in the main product line.

This fork ensures the retention of core functionalities while enabling targeted modifications. Thus, forks facilitate targeted adaptations and exploratory development in response to evolving requirements.

3.2 Data source

Table 1 summarizes the types of development activities that can be supported in general by the selected data sources—issues, commits, and pull requests—without requiring additional advanced analysis or tooling. The goal is to characterize the potential of each source in terms of the information it can provide for feedback extraction. A *Yes* indicates native support for the activity, while ± denotes partial or indirect support [12].

Commits offer valuable insights into code changes and fixes, while issues and merge requests are important for their textual information and tagging capabilities. Therefore, to gain a comprehensive understanding of a project's behavior and evolution, it is essential to combine multiple data sources from version control, project tracking systems, and collaboration workflows.

- *Issues:* They allow for identifying recurring problems, feature requests, and system weaknesses reported by users or

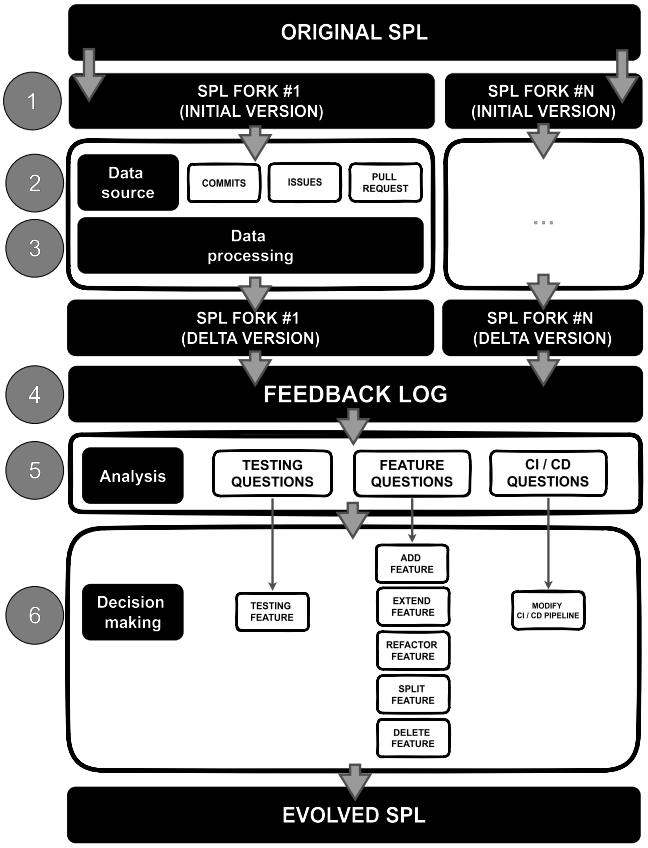

Figure 2: Automatic feedback collection

Data source	Obtain textual information	Files mapping	Files evolution	Identify corrections	Labelling
Issues	Yes	Yes	±	Yes	Yes
Commits	Yes	Yes	Yes	Yes	±
PR	Yes	±	±	Yes	Yes

Table 1: Relation between data sources and supported activity types

fork developers. This source helps detect conflicting functionalities, recognize needs not covered by the original SPL, and extract relevant descriptive tags to categorize changes.

- *Commits:* They reflect the technical evolution of each fork, including modified files, the type of change (addition, deletion, modification), and associated messages. Through commit analysis, it is possible to track the evolution of a feature, associate changes to specific technical decisions such as refactoring or extensions, and measure the frequency and impact of modifications over time.
- *Pull Requests (PR):* They represent integration proposals, collaborative reviews, and validations through automatic pipelines. Their analysis allows for an understanding of the dynamics of developer collaboration, identifying which changes were accepted or rejected and why, and observing the results of automatic tests associated with new functionalities or refactoring.

While the current implementation emphasizes issues, commits, and pull requests, the proposed methodology is extensible to incorporate additional feedback sources. Other data sources—such as code review comments, CI/CD logs, test coverage reports, or runtime telemetry—can also be integrated to enrich the feedback log. The flexibility of the model allows adaptation to different development workflows.

3.3 Data processing

Our proposal defines a traceability mechanism that captures how development activity relates to specific features in the SPL. This is formalized as a ternary relation that captures the association between features, data sources, and semantic descriptors of development activity.

In general, such informational elements could take many forms—e.g., extracted patterns, metadata, or semantic annotations. In our case, we instantiate this dimension using *semantic tags*: textual labels (e.g., `fix`, `mock`, `ci`) assigned to elements from data sources—such as issues, commits, or pull requests—that convey meaningful information about the nature of the change.

The goal is to formalize a structure that captures the relationship between three key dimensions:

- **Feature** — a functional unit defined in the SPL variability model.
- **Data source type** — the origin of the information (e.g., `commit`, `issue`, `pull request`).
- **Tag** — a semantic label describing the nature of the associated activity.

We formalize the traceability relation as a ternary relation:

$$T \subseteq \mathcal{F} \times \mathcal{D} \times \mathcal{L} \tag{1}$$

where:

- $\mathcal{F}$ is the set of features in the SPL,
- $\mathcal{D}$ is the set of data source types,
- $\mathcal{L}$ is the set of semantic tags used in our implementation.

Each tuple $\langle f, d, \ell \rangle \in T$ indicates that feature f is associated with tag ℓ, extracted from data source type d.

This traceability structure supports many-to-many mappings between dimensions: a single feature may be associated with multiple tags and data sources, and conversely, a tag may be used across multiple features and sources. The structure T serves as the analytical foundation for answering feature-centric questions, as developed in Section 3.5.

The mapping process is flexible, adapting to various project structures and technologies. It allows for both manual and automated identification and tagging of features. This phase directly addresses *RQ1*, which explores how development activities from derivative forks can be collected and associated with features. We

systematically capture the relationships between features, data sources, and semantic annotations by organizing repository data into a traceability structure. This setup enables accurate tracking of feature-related activities across different forks. To establish this traceability structure, we implement a three-step mapping process:

3.3.1 Feature identification and location. The process starts by obtaining a list of SPL features. Domain experts can manually define or infer this list from system artifacts such as documentation, variability models, or code structure.

3.3.2 Linking data sources to features. Issues, commits, and pull requests are collected from the project repository. Each data source type is linked to one or more features based on the files it affects. If a data source type (e.g., a commit) modifies files known to implement a given feature, it is associated with the corresponding feature.

3.3.3 Tag assignment. Semantic tags are assigned to each element. This step can be explicit—reusing existing labels from issue trackers or version control systems—or inferred using automatic techniques. For example, tagging can be automated based on predefined categories, helping identify features under development, fixed, or discussed. Additionally, word cloud analysis can be used to visualize the most frequent terms in commit messages, issue descriptions, or documentation, offering insights for tag suggestion or refinement.

3.4 Feedback log

Up to this point, each fork has been processed independently. Feature identification, linkage, and tagging were applied in isolation. The *feedback log* consolidates the results from all forks into a unified, structured dataset that preserves semantic, contextual, and temporal information.

$$\Phi \subseteq \mathcal{F} \times \mathcal{D} \times \mathcal{L} \times \mathcal{K} \times \mathcal{Z} \tag{2}$$

where:

- $\mathcal{F}$ is the set of features,
- $\mathcal{D}$ is the set of data source types,
- $\mathcal{L}$ is the set of semantic tags,
- $\mathcal{K}$ is the set of forks,
- $\mathcal{Z}$ is the set of timestamps.

Each tuple $\langle f, d, \ell, k, z \rangle \in \Phi$ represents a tagged development activity associated with feature f, extracted from data source type d, carrying tag ℓ, and originating from fork k at time z. In our implementation, the informational elements in $\mathcal{L}$ are instantiated as semantic tags that summarize the nature of the activity (e.g., `fix`, `mock`, `ci`).

3.5 Analysis

The analysis focuses on how features evolve across forks and how tagged elements (issues, commits, and pull requests) reflect this evolution. This analysis addresses *RQ2*, which examines how traceability data can support decision-making in testing, feature evolution, and CI/CD workflows within an SPL. By analyzing tagged development artifacts across forks, we extract structured feedback that helps identify testing gaps, assess the evolution of specific features, and understand their impact on integration and deployment processes. Using the structure defined in Structure 2, we can query the dataset to identify patterns related to:

3.5.1 Testing questions. Testing is crucial to ensure feature reliability and maintainability. The following aspects are analyzed:

- *Which features require additional testing?* Identifying features with limited test coverage helps prioritize test development.
- *Which features require refactoring to improve testability?* Some features may be tightly coupled or have complex dependencies, making testing difficult.
- *Which integration tests are most needed?* Determining critical feature interactions that require end-to-end validation.
- *Which tests rely heavily on mocks?* Analyzing the excessive use of mocks can reveal coupling between features, suggesting potential redesigns.

3.5.2 Feature questions. Feature evolution across forks provides insights into how the SPL can be improved. The following aspects are examined:

- *Which features are extended with new models?* Identifying extensions that enhance existing features.
- *Which features should be evolved (added, removed, refactored)?* Evaluating necessary changes based on usage trends and feedback.
- *Which similar features have been implemented across different forks?* Detecting redundant implementations of the same functionality across forks to unify efforts.

3.5.3 CI / CD questions. Feature changes can introduce issues in configuration and CI/CD workflows. The following questions are considered:

- *Which features cause configuration errors?* Analyzing configuration issues introduced by feature modifications.
- *Which configurations trigger new features in the framework?* Understanding how certain configurations lead to the activation of specific features.
- *Which features fail more often in production?* Identifying features with high failure rates in real-world deployments.
- *Which features behave differently in production compared to development?* Investigating discrepancies between development and production environments.
- *Which features introduce new CI/CD workflows?* Understanding how feature modifications impact deployment processes.

3.6 Decision-making

Based on the analysis's results, application engineers make informed decisions to evolve the SPL. These decisions are grouped into three main categories, aligned with the types of questions addressed in the previous section.

3.6.1 Testing decisions. Testing-related analysis helps identify gaps and weaknesses in feature validation. As a result, engineers may:

- *Test feature:* Improve or create test cases for features with limited coverage or recurring issues.

3.6.2 Feature evolution decisions. Feature-centric analysis reveals opportunities to improve the variability model and the overall architecture. Possible actions include:

- *Add feature:* Introduce new features identified in forks.
- *Extend feature:* Integrate enhancements observed in derivative developments.
- *Refactor feature:* Restructure features for better modularity or maintainability.
- *Split feature:* Decompose large or coupled features into smaller units.
- *Delete feature:* Remove redundant or obsolete features.
- *Modify architecture:* Apply structural changes motivated by recurring design or integration issues.

3.6.3 CI/CD and configuration decisions. Findings related to continuous integration and deployment may lead to adjustments in development workflows:

- *Modify CI/CD pipeline:* Adapt or extend the automation process to accommodate new feature behaviors or resolve integration failures.

All these actions are applied traceably and consistently, ensuring that the original SPL evolves based on evidence gathered from parallel developments.

4 Evaluation

This section assesses both the feasibility and scalability of the proposed feature-centric feedback analysis method in the context of SPLs. We applied the proposed methodology to empirical data extracted from GitHub repositories, constructing traceability maps and feedback logs for analysis. The resulting analysis highlights actionable insights, including unmet testing requirements, candidate features for refactoring, evidence of feature evolution, and trends in CI/CD workflow adaptations. Results are obtained from an individual fork and a larger dataset of forks, showing that the approach can handle both local and global SPL scenarios. Insights are presented through quantitative summaries and visual representations.

4.1 Proof of concept

To assess the viability and effectiveness of our approach in a controlled setting, we initially performed a proof-of-concept study on a single fork of the target SPL[1]. This preliminary evaluation established the foundation for subsequent large-scale analysis involving multiple forks.

Unlike the full evaluation presented in Section 4.2, this case study focuses solely on one development line and excludes the construction of the *feedback log*. Instead, it centers on building a traceability map of the form:

$$\langle \text{feature, data source type, tag} \rangle$$

4.1.1 Data collection and storage. We extracted all commits, issues, and pull requests from the selected fork using the GitHub API, resulting in:

- **183** commits
- **41** issues
- **19** pull requests

The extracted data was serialized into structured JSON format to enable systematic processing and analysis.

4.1.2 Traceability map construction. Feature identification was automated by leveraging the repository's modular structure, specifically by parsing directory names under `app/modules` [2]. Then, each commit message, issue title, and pull request description was analyzed using keyword-based heuristics to detect semantic tags such as `test`, `fix`, or `extension`. These tags were mapped to predefined analysis categories.

Finally, traceability tuples were constructed by linking each tag to the affected feature(s), based on modified file paths or GitHub labels. We extracted **56 traceability tuples** from this fork.

4.1.3 Analysis results. Using the resulting traceability map, we answered the research questions outlined in Section 3.5. The following summarizes key findings derived from the traceability analysis:

Testing-related insights.

- *Which features require additional testing?* Tags `testing`, `fix`, and `error` were frequently associated with `feature7`, `feature3`, and `feature10`.
- *Which features rely heavily on mocks?* The recurrence of `mock` tags associated with `feature7` and `feature4` may indicate design limitations or a high degree of inter-feature coupling.
- *Which features need refactoring?* `feature3` and `feature7` were repeatedly tagged with `refactor`, highlighting testability and maintainability concerns.

Feature evolution insights.

- *Which features are being extended?* The most extended modules were `feature7` and `feature11`, with multiple commits tagged as `extension`.
- *Which features show signs of obsolescence?* `feature8` and `feature4` were mentioned in issues tagged `obsolete`, suggesting that they might be candidates for removal.
- *Which features vary across forks?* Evidence of feature divergence was identified, even within the scope of a single fork, particularly in `feature6` and `feature28`, where similar functionality was implemented via different strategies.

CI/CD-related insights.

- *Which features caused configuration issues?* Tags such as `config` and `setup` were frequently linked to `feature7`, `feature23`, and `feature17`, indicating deployment or pipeline complexity.
- *Which features introduced new models or configurations?* `feature7`, `feature3`, and `feature11` were the primary contributors to new configuration structures and models within the fork.

These findings indicate that the proposed approach can yield meaningful insights even when applied to an individual forked development.

[1] Repository: XXX. The repository has been anonymized and will be made public if this paper is accepted.

[2] The feature names in this evaluation have all been modified to *featureX* style for anonymization purposes.

4.2 Evaluation with multiple forks

To evaluate the scalability and global reasoning capabilities of our approach, we applied the full methodology, including feedback log construction, to a dataset comprising 25 forks of the same SPL.

Each fork was independently processed in three steps: (1) extracting raw data from GitHub (commits, issues, pull requests), (2) identifying features from the project structure (`app/modules`), and (3) building traceability tuples. These tuples were then aggregated into a unified *feedback log*, enabling cross-fork, feature-centric, and time-aware analysis.

A total of **10,163** feedback entries were collected, covering **38 unique features** across all forks.

4.2.1 Global analysis results. Based on this consolidated dataset, we performed global analysis to answer the questions defined in Section 3.5. The most relevant insights are summarized below:

- *Testing questions:*
  - Features such as `feature7`, `feature3`, and `feature12` show the highest need for additional testing, with more than 700 combined entries tagged as `testing`, `fix`, or `error`.
  - `feature12` and `feature7` also exhibit high `mock` usage, indicating complex dependencies and potential design limitations.
  - `feature7`, `feature3`, and `feature10` appear in numerous entries tagged as `refactor`, suggesting opportunities to improve testability.
- *Feature questions:*
  - `feature7`, `feature3`, and `feature12` are the most frequently extended features, each with over 100 occurrences tagged as `extension`.
  - `feature7` also leads in entries tagged `refactor` and `obsolete`, highlighting both active evolution and technical debt.
  - Features like `feature20`, `feature19`, and `feature17` appear in multiple forks with varying implementations.
- *CI/CD questions:*
  - Features such as `feature12`, `feature7`, and `feature23` are linked to configuration errors, based on issues tagged `config`.
  - `feature7`, `feature11`, and `feature3` frequently introduce new models and configuration patterns in CI/CD workflows.
  - `feature7` also ranks highest in combined `testing`, `config`, and `model` tags, reflecting its central role in development activity.

The results suggest that the proposed method is capable of scaling to real-world, multi-fork SPL scenarios and provides structured, actionable insights to support SPL maintenance and evolution.

4.2.2 Visual feedback analysis. To complement the textual analysis of the feedback log, we generated a log-scaled semantic activity heatmap, as shown in Figure 4. It shows the relationship between features and semantic tags. Each row is a feature. Each column is a tag. Darker cells indicate more activity. We use a logarithmic scale to enhance contrast. This visualization provides a high-level overview of development patterns across all forks.

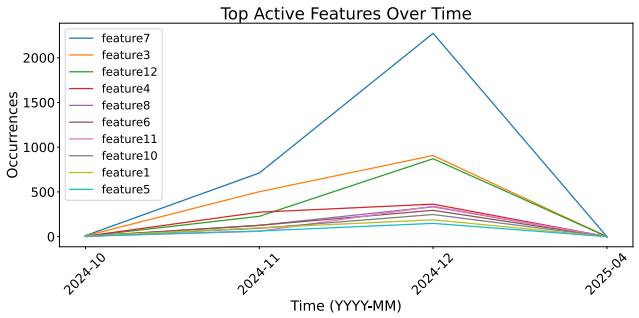

Figure 3: Temporal evolution of the most active features across forks.

This heatmap is constructed directly from the feedback log entries of the form $\langle f, d, \ell, k, z \rangle$, where each row corresponds to a feature and each column to a semantic tag (e.g., `testing`, `fix`, `mock`, etc.). The intensity of each cell indicates the number of times that feature appeared with that tag across all forks. To improve visual contrast and emphasize both frequent and infrequent activity, we applied a base-10 logarithmic transformation to the values.

This visualization enables:

- Identifying features with strong semantic diversity (i.e., involved in many types of development activity).
- Detecting features primarily associated with specific concerns such as testing, modeling, or refactoring.
- Spotting semantically inactive features, which may indicate stability or neglect.

The heatmap provides a concise and feature-centric summary of SPL activity across forks, helping maintainers and researchers prioritize integration, testing, or refactoring tasks.

Figure 3 shows the temporal evolution of the top 10 most active features based on monthly occurrences in the feedback log. Each line represents a feature's activity over time. One feature (`feature7`) dominates the others, demonstrating how our approach helps identify focal points of development. This visualization assists maintainers in identifying high-impact features and allocating improvement efforts based on activity density and temporal trends. A feature is considered active if it accumulates many feedback log entries, regardless of the semantic tag or data source. Frequent appearances suggest that the feature is under development, maintenance, or evolution across forks.

5 Threats to validity

We identify the following potential threats to the validity of our proposal, along with mitigation strategies:

- *Internal validity: imprecise mapping between files and features.* The assumption that modified files are directly associated with specific features may not always hold, especially in systems with poor modularization. *Mitigation:* Manually validate a sample of mappings or use historical and dependency analysis to strengthen associations.

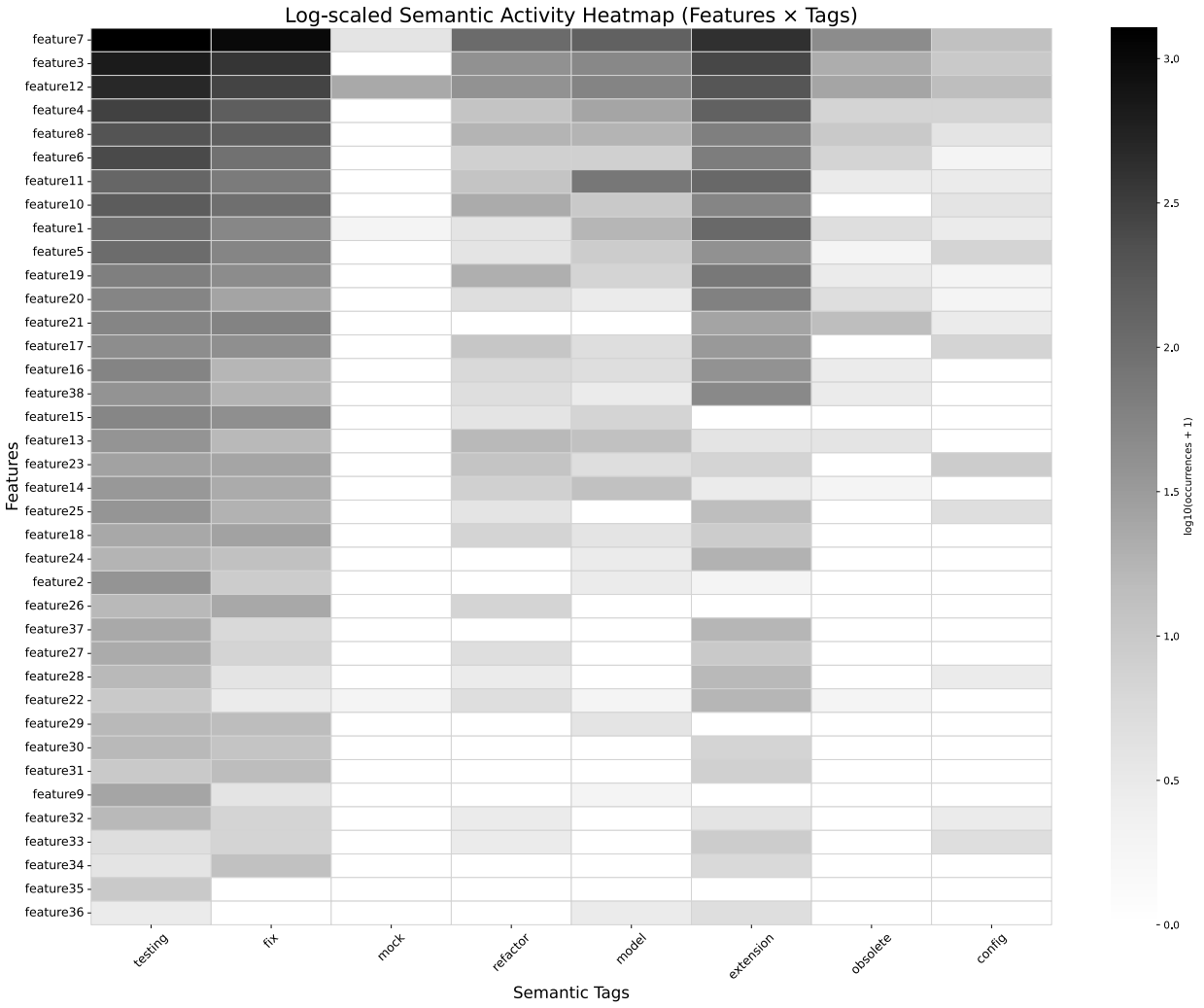

Figure 4: Log-scaled Semantic Activity Heatmap across features and tags.

- *Construct validity: incomplete or inconsistent tagging.* Tags extracted from issues, commits, or pull requests may be absent, ambiguous, or inconsistent. *Mitigation:* Combine automatic tagging with manual curation, or train supervised models using a curated dataset.
- *External validity: limited generalizability.* The approach may be tailored to specific types of repositories (e.g., GitHub-based, open-source, well-documented projects). *Mitigation:* Apply the method to diverse repositories (different languages, sizes, structures) to assess generalizability.
- *Conclusion validity: biased interpretation of data.* Decisions derived from the analysis may be influenced by subjective interpretation or coincidental correlations. *Mitigation:* Define objective criteria for actions and maintain traceability between data points and decisions.
- *Ecological validity: limited applicability in real-world workflows.* The proposed process might be difficult to adopt in industrial settings or existing development workflows. *Mitigation:* Design the approach as a modular and integrable tool compatible with current platforms and pipelines.

6 Discussion and limitations

This work has introduced a methodology for the automated collection of development-related information from SPL forks. A traceability structure was designed to associate development activities with specific features. The evaluation shows how this information helps identify relevant aspects for the evolution of features, tests, and CI/CD. However, the practical applicability depends on organizational dynamics, fork motivations, and alignment with SPL workflows.

Although our proposal focuses on the technical aspects of feedback collection from forks, its successful adoption in industrial settings depends heavily on organizational readiness. Prior studies

have identified several obstacles to adopting SPL practices, including organizational inertia, lack of tooling support, and resistance to formal variability modeling [1, 6].

Similarly, adopting a feedback-driven evolution process, as proposed here, requires organizations to:

- Ensure alignment between distributed teams working on forks and the central SPL team.
- Establish governance structures to evaluate and accept feedback contributions from forks.
- Encourage practices such as structured tagging and disciplined use of version control metadata, which are prerequisites for effective traceability.

Even technically robust processes may yield limited value if not supported by adequate organizational practices and governance structures. Forks differ in origin, purpose, and relevance. Some arise from technical needs, while others reflect organizational factors [18, 30]. Understanding fork motivations is key to interpreting feedback. Long-term divergent forks may be less relevant, while short-term forks can provide valuable insights. Although our approach does not distinguish motivations, future work could apply heuristics to prioritize reintegration.

Feedback-driven SPL evolution, as proposed in this work, can complement existing SPL workflows by (1) guiding variability model evolution in domain engineering (e.g., refactoring or removing features), (2) exposing configuration and implementation issues in application engineering, and (3) surfacing feature-level failures in DevOps for CI/CD integration. Our methodology must be integrated as a modular analysis tool that complements existing product line engineering (PLE) environments to enable this. Integration points may include feature modeling tools, configuration management systems, and release planning workflows [6].

This work does not encompass the reintegration of collected feedback into the SPL architecture, which remains as future work. While we describe the format and structure of the feedback log, converting this data into architectural or variability model changes is left for future work. The approach assumes feature definitions exist and are stable. In real cases, features may evolve or be reinterpreted. The quality of insights depends on metadata consistency and developer practices.

A critical limitation is the dependency on semantically rich commits and descriptive pull requests, which are not guaranteed. Inconsistent or insufficient version control metadata may compromise traceability and feedback accuracy. Addressing this may require guidelines, templates, or validation. The proposal assumes a modular architecture linking artifacts to features. In practice, monolithic systems may dilute these links, requiring static analysis or developer input to improve accuracy.

7 Related work

Medeiros et al. [28] propose studying feedback at the platform level, enabling automatic tracking code generation and preserving platform/variant separation. Their study links implicit feedback to benefits such as feature model analysis, dependency detection, and test prioritization. In contrast, our approach analyzes external forks, focusing on indirect signals like commits and feature evolution. Unlike Medeiros et al., we rely on structural changes rather than usage data.

Shurui et al. introduce *Infox*, a methodology and automated tool designed to identify and summarize features implemented in software project forks through source code analysis, community detection techniques, and information retrieval [43]. *Infox* creates dependency graphs using static source code analysis and groups related code fragments using community detection. It tags these sets with keywords derived from commit messages, code, and comments. Evaluation demonstrated a median accuracy of 90%. Unlike *Infox*, our proposal already has localized features and studies the evolution of their development.

Oscar Diaz et al. [15] propose to move implicit feedback collection from variants to the platform in SPLs, using generative mechanisms. It identifies SPL activities that can benefit from this approach and assesses its feasibility through an exploratory study with practitioners. This implicit feedback is studied in the use of the products. In our proposal, we focus on collecting feedback in the development of parallel SPLs, not on using a single product.

8 Conclusions and future work

This paper has presented a structured method for automating the collection of technical feedback in forked SPLs. The approach formalizes a traceability structure that associates development artifacts with SPL features via semantic tags. This supports feature-centric analysis across testing, evolution, and CI/CD. The feedback log facilitates SPL evolution by aggregating insights from parallel development activities.

A notable strength of the method lies in its capacity to reestablish traceable connections between decentralized fork activity and the central SPL variability model. By consolidating activity across forks, maintainers can coherently regain visibility and guide evolution. Its modular design and extensible tagging mechanism enable adaptability to diverse project structures, while producing interpretable outputs suitable for supporting maintenance and evolution decisions.

Nonetheless, the approach presents several limitations that warrant further attention. The mapping assumes a clear file-feature relationship, which may not hold in poorly modularized systems. Furthermore, the effectiveness of semantic tagging is contingent on the quality and consistency of metadata, which is often heterogeneous across repositories.

Future work will explore enhanced feature inference techniques based on call graph analysis, dependency mining, and historical code evolution. Another direction involves training supervised learning models on curated datasets to improve the accuracy and consistency of semantic tag assignment.

In summary, development activity within forked repositories constitutes a valuable and largely untapped source of structured feedback. Making this feedback explicit and traceable helps maintainers benefit from distributed innovation. The proposed approach lays the groundwork for automated, evidence-based tools that support the evolution and maintenance of SPL ecosystems.

Material

Supplementary material to this article will be made public after acceptance. It contains information that could reveal the identity of the authors, and it is not possible to anonymise it without compromising its integrity.

References

[1] 2001. *Software product lines: practices and patterns.* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Development Apache, Audris Mockus, Roy Fielding, and James Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology* 11 (09 2002). doi:10.1145/567793.567795

[3] Jakob Axelsson. 2009. Evolutionary architecting of embedded automotive product lines: An industrial case study. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*. 101–110. doi:10.1109/WICSA.2009.5290796

[4] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, Henk Obbink, and Klaus Pohl. 2001. Variability Issues in Software Product Lines. 13–21. doi:10.1007/3-540-47833-7_3

[5] John Businge, Moses Openja, Sarah Nadi, and Thorsten Berger. 2022. Reuse and maintenance practices among divergent forks in three software ecosystems. *Empirical Software Engineering* 27, 2 (2022), 54. doi:10.1007/s10664-021-10078-2

[6] Cagatay Catal. 2009. Barriers to the adoption of software product line engineering. *SIGSOFT Softw. Eng. Notes* 34, 6 (Dec. 2009), 1–4. doi:10.1145/1640162.1640164

[7] Ana Eva Chacón-Luna, Antonio Manuel Gutiérrez, José A. Galindo, and David Benavides. 2020. Empirical software product line engineering: A systematic literature review. *Information and Software Technology* 128 (2020), 106389. doi:10.1016/j.infsof.2020.106389

[8] Katharine Chen, Maria Toro-Moreno, and Arvind Subramaniam. 2025. GitHub is an effective platform for collaborative and reproducible laboratory research. *ArXiv* (02 2025).

[9] Siyue Chen, Loek Cleophas, Sandro Schulze, and Jacob Krüger. 2024. Use the Forks, Look! Visualizations for Exploring Fork Ecosystems. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 993–1004. doi:10.1109/SANER60148.2024.00107

[10] Sekhar Chittala. 2024. Enhancing Developer Productivity Through Automated CI/CD Pipelines: A Comprehensive Analysis. *International Journal of Computer Engineering Technology* 15 (10 2024), 882–891. doi:10.5281/zenodo.13929524

[11] Zadia Codabux, Fatemeh Fard, Roberto Verdecchia, Fabio Palomba, Dario Nucci, and Gilberto Recupito. 2025. Teaching Mining Software Repositories. doi:10.48550/arXiv.2501.01903

[12] Cleidson R. B. de Souza, Emilie Ma, Jesse Wong, Dongwook Yoon, and Ivan Beschastnikh. 2024. Revealing Software Development Work Patterns with PR-Issue Graph Topologies. *Proc. ACM Softw. Eng.* 1, FSE, Article 106 (July 2024), 22 pages. doi:10.1145/3660813

[13] Oscar Díaz, Leticia Montalvillo, Raul Medeiros, Maider Azanza, and Thomas Fogdal. 2022. Visualizing the customization endeavor in product-based-evolving software product lines: a case of action design research. *Empirical Software Engineering* 27, 3 (2022), 75.

[14] Jane Doe, Alex Johnson, and Adebis Samuel. 2025. Introduction to GitHub Actions: Building and Automating Workflows. (03 2025).

[15] Oscar Díaz, Raul Medeiros, and Mustafa Al-Hajjaji. 2024. How can feature usage be tracked across product variants? Implicit Feedback in Software Product Lines. *Journal of Systems and Software* 211 (2024), 112013. doi:10.1016/j.jss.2024.112013

[16] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolás Serrano. 2025. DevOps 2.0. *IEEE Software* 42 (03 2025), 24–32. doi:10.1109/MS.2025.3525768

[17] Floris Erich, Chintan Amrit, and Maya Daneva. 2014. Report: DevOps Literature Review. (10 2014). doi:10.13140/2.1.5125.1201

[18] Neil A. Ernst, Steve Easterbrook, and John Mylopoulos. 2010. Code forking in open-source software: a requirements perspective. arXiv:1004.2889 [cs.SE] https://arxiv.org/abs/1004.2889

[19] Aleksander Fabijan, Pavel Dmitriev, Helena Holmström Olsson, and Jan Bosch. 2017. The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 770–780. doi:10.1109/ICSE.2017.76

[20] Dimitris Giamos, Olivier Doucet, and Pierre-Majorique Léger and. 2024. Continuous Performance Feedback: Investigating the Effects of Feedback Content and Feedback Sources on Performance, Motivation to Improve Performance and Task Engagement. *Journal of Organizational Behavior Management* 44, 3 (2024), 194–213. doi:10.1080/01608061.2023.2238029 arXiv:https://doi.org/10.1080/01608061.2023.2238029

[21] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. 2012. *Traceability Fundamentals.* Springer London, London, 3–22. doi:10.1007/978-1-4471-2239-5_1

[22] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. 2012. Evolving Delta-Oriented Software Product Line Architectures, Vol. 7539. 183–208. doi:10.1007/978-3-642-34059-8_10

[23] Ahmed E. Hassan. 2008. The road ahead for Mining Software Repositories. In *2008 Frontiers of Software Maintenance*. 48–57. doi:10.1109/FOSM.2008.4659248

[24] Yash Jani. 2023. Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development. *International Journal of Science and Research (IJSR)* 12 (06 2023), 2984–2987. doi:10.21275/SR24716120535

[25] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 92–101. doi:10.1145/2597073.2597074

[26] Christoph Knieke, Andreas Rausch, Mirco Schindler, Arthur Strasser, and Martin Vogel. 2022. Managed Evolution of Automotive Software Product Line Architectures: A Systematic Literature Study. *Electronics* 11, 12 (2022). doi:10.3390/electronics11121860

[27] Walid Maalej, Maelnaz Nayebi, Timo Johann, and Guenther Ruhe. 2015. Toward Data-Driven Requirements Engineering. *IEEE Software* 33 (01 2015), 48–56. doi:10.1109/MS.2015.153

[28] Raul Medeiros, Oscar Díaz, and David Benavides. 2023. Unleashing the Power of Implicit Feedback in Software Product Lines: Benefits Ahead. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 113–121. doi:10.1145/3624007.3624058

[29] Andreas Metzger and Klaus Pohl. 2014. Software product line engineering and variability management: achievements and challenges. In *Future of Software Engineering Proceedings* (Hyderabad, India) *(FOSE 2014)*. Association for Computing Machinery, New York, NY, USA, 70–84. doi:10.1145/2593882.2593888

[30] Linus Nyman. 2015. *Understanding Code Forking in Open Source Software: An examination of code forking, its effect on open source software, and how it is viewed and practiced by developers.* Ph. D. Dissertation. Hanken School of Economics.

[31] Jakub Perdek and Valentino Vranic. 2025. Fully Automated Software Product Line Evolution With Diverse Artifacts. *IEEE Access* 13 (2025), 27325 – 27358. doi:10.1109/ACCESS.2025.3539868 Cited by: 0; All Open Access, Gold Open Access.

[32] Klaus Pohl, Gunther Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer.

[33] Mohammad Masudur Rahman and Chanchal K. Roy. 2014. An insight into the pull requests of GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 364–367. doi:10.1145/2597073.2597121

[34] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference - Volume 1* (Salvador, Brazil) *(SPLC '12)*. Association for Computing Machinery, New York, NY, USA, 156–160. doi:10.1145/2362536.2362558

[35] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. doi:10.1109/ACCESS.2017.2685629

[36] Venkata Mohit Tamanampudi. 2024. End-to-End ML-Driven Feedback Loops in DevOps Pipelines. *World Journal of Advanced Engineering Technology and Sciences* 13 (09 2024), 340–354. doi:10.30574/wjaets.2024.13.1.0424

[37] Mohammad Tanhaei, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. 2016. Automating feature model refactoring: A Model transformation approach. *Information and Software Technology* 80 (2016), 138–157. doi:10.1016/j.infsof.2016.08.011

[38] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85. doi:10.1016/j.scico.2012.06.002 Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[39] Andrii Tkalich, Eduards Klotins, Trygve Sporsem, et al. 2025. User feedback in continuous software engineering: revealing the state-of-practice. *Empirical Software Engineering* 30, 79 (2025). doi:10.1007/s10664-024-10557-2

[40] Anastasiia Tkalich, Eriks Klotins, Tor Sporsem, Viktoria Stray, Nils Brede Moe, and Astri Barbala. 2024. User Feedback in Continuous Software Engineering: Revealing the State-of-Practice. arXiv:2410.07459 [cs.SE] https://arxiv.org/abs/2410.07459

[41] Mariot Tsitoara. 2020. *Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer.* doi:10.1007/978-1-4842-5313-7

[42] Melina Vidoni. 2021. A systematic process for Mining Software Repositories: Results from a systematic literature review. *Information and Software Technology* (12 2021), 106791. doi:10.1016/j.infsof.2021.106791

[43] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying features in forks. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/3180155.3180205

[44] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How has forking changed in the last 20 years? a study of hard forks on GitHub. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 445–456. doi:10.1145/3377811.3380412