IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 3 June 2025, accepted 27 June 2025, date of publication 7 July 2025, date of current version 11 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3586537

RESEARCH ARTICLE

Branch Drift: A Visually Explainable Metric for Consistency Monitoring in Collaborative Software Development

KARL KEGEL, SEBASTIAN GÖTZ, AND UWE AßMANN
Faculty of Computer Science, Chair of Software Technology, Dresden University of Technology, 01069 Dresden, Germany
Corresponding author: Karl Kegel (karl.kegel@tu-dresden.de)

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)–SFB 1608-501798263.

ABSTRACT Modern software engineering uses branch-based workflows for collaborative development on a shared codebase. A branch may represent a task or feature within the responsibilities of a specific developer or team. When branches are merged, unanticipated conflicts occur frequently. Their resolution is time-consuming and costly. Depending on the development process, a branch exists for a few hours or up to several months. With increasing time and many branches, keeping track of potential merge conflicts becomes difficult. Typically, developers are unaware of these conflicts until the branches are merged. Issues solvable through joint reviews, discussion, and collaboration are therefore identified and resolved late. It is evident that the effort required to resolve merge conflicts poses an organizational technical debt. Project managers and team leaders must be able to track, analyze, and interpret the merge conflict potential to call for timely actions. Therefore, a metric is required. This work transfers the *drift* metric for inconsistency analysis from model-driven software engineering to branch-based software development. The resulting *branch-drift* metric summarizes the merge conflict potential of a development project in a numeric value. Its calculation is configurable to fit any project's requirements. The branch-drift is based on an intuitive and explainable geometric representation, easing the analysis of conflict patterns and outliers. We evaluate the branch-drift by quantitatively analyzing and discussing the drift values, trends, and patterns for 21 large-scale open-source repositories over three months. The results of this work encourage and enable the continuous and configurable branch-drift calculation for arbitrary Git repositories. The developed tools and gathered data are publicly available.

INDEX TERMS Software engineering, collaborative development, metrics, technical debt, quality assurance.

I. INTRODUCTION

State-of-the-art software development workflows use branch-based *version control systems (VCS)* as the backbone for collaborative development. The technical concepts of branch-based VCS have been around for decades [1] and are well-adopted by today's software engineering processes. A branch contains a temporary copy of the codebase for modification purposes. After receiving several changes, i.e., commits, the branch may be merged back to its origin or is discarded if the changes turn out to be not worthwhile to merge. Collaborative software development without a VCS like *Git*[1] is nowadays hard to imagine. Git-based web platforms, such as *Github*[2] or *Gitlab*[3] provide remote hosting of Git repositories along with additional features, including issue tracking, communication, and project management tools.

Branching, i.e., branch-based development, has been recognized as a crucial contributor to the success of software development projects and their quality [2]. An extensive

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

[1]https://git-scm.com/
[2]https://github.com/
[3]https://about.gitlab.com/

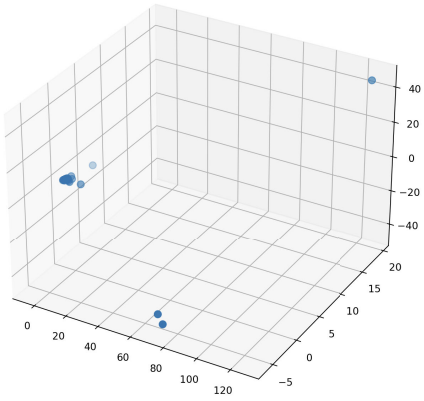

FIGURE 1. Example of a real repository's drift plot. The figure shows the plot ofthe *julia* repository's analysis result (26. Feb '24) from the evaluation in sec. VI. The Euclidean distance approximates the number of files with merge conflicts.

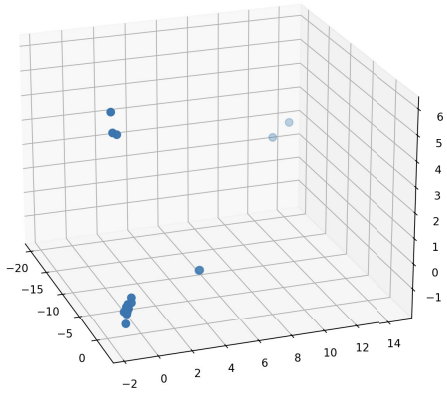

FIGURE 2. Example of a real repository's drift plot. The figure shows the plot of the *electron* repository's analysis result (26. Feb '24) from the evaluation in sec. VI. The Euclidean distance approximates the number of files with merge conflicts.

analysis of open-source repositories showed that branches are widely used in Git projects [3]. The survey by César et al. presents a multitude of different branch-based workflows [4]. A well-known representative is the *Feature-Branch* workflow, which motivates the development of new features in dedicated branches with short lifetimes. These branches are called *feature branches*. After a developer finalizes the implementation of a feature in their feature branch, the branch is merged back into the main branch. However, this idea is not exclusive to the feature branch workflow; other workflows also employ variations of this approach [4].

Whenever a merge is performed, *merge conflicts* can occur. A merge conflict is caused by a pair of competing changes to the same code fragment. Resolving merge conflicts is a time-consuming task and an organizational challenge [5]. Merge conflicts occur frequently [6], [7]. Although ''smart'' merge strategies can reduce the number of merge conflicts [8], [9], a complete prevention is generally impossible. Merge conflicts manifest during the merge. However, their cause lies in the past development interval. We call a merge conflict not yet manifested but already introduced in a code fragment a *potential conflict*. Later not-manifestation can be caused by active or accidental removal of the conflicting code fragment, introduction of the same change in both branches or deletion of the branch containing the problematic code fragment. In addition, two branches having a potential conflict may never merge. A potential conflict may lie *dormant* if the involved fragments are located in long-living branches.

We are aware of two general approaches for detecting conflicts early on. One approach focuses on machine learning methods to predict conflicts based on the current codebase, past development data, and additional artifacts such as the commit history [10], [11]. The benefit of this approach is that it can predict conflicts and approximate their difficulty. The downside is that these approaches typically work probabilistically, i.e., uncertainty always remains. In addition, the algorithms used must be trained on a suitable dataset that fits the characteristics of the project. The second approach, which is conceptually simpler, is to detect merge conflicts by actually attempting to merge [12]. By doing so, the merge tool retrieves the actual conflicts and processes them further. This approach does not require any training data and works on any project. However, performing merges, i.e., code differencing operations, is resource intensive.

We call the expected number and severity of conflicts during a future merge the *merge-conflict potential*. Merge-conflict potential qualifies as *technical debt* [13], [14] within a development project. Merge conflicts are not only relevant as a development debt but also as an organizational debt due to the organizational aspects of their resolution [5]. Both preventing and resolving merge conflicts require the joint effort and communication of all developers involved.

A. PROBLEM STATEMENT

We identify merge-conflict potential as a form of technical debt. Effectively handling technical debt requires quantifying it in a metric. Continuous observation of a risk factor leads to awareness and timely countermeasures. The analysis of Zou et al. [3] indicates that over 80% of openly available repositories use more than one branch for development. The study by Ghiotto et al. [6] shows that 2.6% of merges fail over a broad range of repositories. Although this number seems low, it is still a significant percentage for a task expected to ''just work'' out of the box. Consequently, we assess a metric for merge-conflict potential as a valuable asset for software development and -management.

A metric for merge conflict potential in software development is not yet established. The recent work by Kegel et al. introduces a *drift metric* [15] as a measure of conflict potential regarding model variations in model-driven engineering. The metric is motivated by *Concept Drift*, a measure of conceptual divergence known from machine learning [16], [17]. The authors motivate the usage of the metric in software development projects and provide means for its calculation.

The main benefit of the drift metric from Kegel et al. is its explainable geometric intermediate representation called *drift plot*. This visualization eases the interpretation of the metric's values and the detection of outliers. For example, Figures 1 and 2 show two drift plots generated from the analysis of real-world repositories. Figure 1 shows the drift plot of the *julia* repository on 26. Feb '24 and 2 shows the drift plot of the *electron* repository on the same date. The plots result from a multidimensional embedding of the repository's branches as points in a unitless 3D space. The distance between the branch points is proportional to the number of merge conflicts between the branches. Both plots show visible clusters of branches and outliers. Clusters are formed by branches that have a low number of merge conflicts compared to the average number of merge conflicts. Multiple clusters also indicate that subsets of branches most likely have the same large merge conflicts in between. As a next step, the development team can investigate the conflicts causing the formation of these structures. In addition, the drift value can be used to track the repository's overall conflict potential over time. However, we still lack a thorough evaluation of the approach using large, real-world repositories observed over a long period to see. Otherwise, one cannot conclude the metric's usefulness nor the amount of effort (computation) required to obtain it for large-scale repositories.

B. APPROACH AND RESEARCH QUESTION

This work aims to establish a metric for merge-conflict potential in software development repositories. Therefore, this work builds upon the drift metric from Kegel et al. [15]. We transfer the conceptual idea of drift from modeling to *branch-drift* in software development. This process requires a solid empirical evaluation of the metric's behavior in real-world repositories over an extended period. Therefore, we conduct an empirical assessment of branch-drift using real-world repository data collected over three months. We interpret the metric's values, trends, and patterns qualitatively and quantitatively. Furthermore, we investigate the technical feasibility of the metric's calculation for large repositories by experimentally studying the influence of different configuration parameters on runtime and metric behavior.

This work aims to answer the following research questions:

RQ1 Is branch-drift a suitable metric for surveilling the merge-conflict potential as a form of inconsistency in branch-based software development projects?

- RQ1.1 Does branch-drift show both coinciding- and not-coinciding hotspots with repository activity metrics?
- RQ1.2 Which underlying pairwise conflict distance (conflict -lines, -occurrences, -files) provides the best results for branch-drift computation?

RQ2 Is the time to compute the branch-drift feasible for large repositories in a typical development workflow?

We approve RQ1 (suitability) if (1) branch-drift occasionally shows coinciding hotspots with other activity metrics, which indicates that it behaves as intuitively expected; (2) branch-drift shows hotspots that do not coincide with- and are not explainable by other activity metrics. Consequently, branch-drift provides additional information that is not covered by other metrics. Otherwise, the metric is obsolete as the same insights can be derived by looking at existing metrics.

In the context of this work, we define explainable behavior as the quantitative or qualitative support of observations based on publicly available metrics, meta-information, and common software development principles. In particular, we consider whether hotspots (high increase or decrease in a short period) in the branch-drift metric are explainable by coinciding hotspots in the repository's activity metrics.

We approve RQ2 if the branch-drift computation's resource consumption— particularly runtime—lies within the magnitude of a few seconds to minutes for large repositories. This time span is feasible, as the drift metric is intended to be computed a few times daily, e.g., in combination with a nightly build or a build on demand.

Overall, this work does not claim or aim to show the suitability of branch-drift for all software development projects. It focuses solely on repositories with feature-branch workflows. However, we do not exclude the possibility that branch-drift (or a modification) is also suitable for other workflows and use cases.

C. CONTRIBUTION

This work contributes the conceptual transfer of the drift metric from model-driven engineering to the branch-drift metric for branch-based software development. Within this process, we provide an analysis of the requirements and challenges for the metric's calculation. We provide full tool support for the branch-drift metric calculation. Therefore, we re-engineered and extended the *Driftool* [15] to provide full-fledged software support for branch-drift computation. Lastly, this work provides an empirical evaluation of the branch-drift metric for 21 large real-world repositories over three months. All tools and data used and presented in this work are made openly available (see Sect. IX). Thus, our evaluation data can be used for further research and validation.

D. METHOD

We start this work with an overview of related work and background research in Section II. In Section III, we present the foundations of the drift metric as introduced by Kegel et al. for software modeling. The concepts presented are then transferred and discussed in the context of branch-based software development in Section IV. Section V presents the *Driftool* as the central tool for *branch-drift* computation. We evaluate our concept and tooling in Sections VI and VII. We close the evaluation by discussing of potential threads to validity in Section VIII. Our supplementary material is

disclosed in Section IX. Finally, we conclude and summarize our work in Section X.

II. BACKGROUND AND RELATED WORK

The related work spans a wide field of research, including consistency in software and systems engineering, development processes, metrics, and collaborative software development. This section will briefly survey important related works regarding consistency, metrics in software engineering, and software merging.

A. CONSISTENCY

Consistency is a relevant concern in software engineering. Recently, consistency has been understood as one of the key factors for the success of large engineering projects. However, consistency is by no means a well-defined term. Current research, therefore, aims to establish unified notions and frameworks for consistency as envisioned by Feichtinger et al. [18]. As there is no consensus on what the terms *consistency* and *inconsistency* mean, the terms have to be used carefully and context-dependently.

This approach of this work is based on the notion of merge consistency, i.e., two things are consistent if there is a conflict-free merge of them. However, this notion might not be sufficient for all cases. For example, a merge might be conflict-free, but the resulting code might still not behave correctly. Wuensche et al. describe conflicts that do not manifest as faulty code but as faulty behavior due to misaligned changes as *higher-order merge conflicts* [19]. Another example is the consistency problem of *Clone-and-Own* workflows in software product-line engineering, as investigated by Tinnes et al. [20]. There, copies of a theoretically common codebase are modified and maintained as independent products. Using a merge-based measure might not help, as the codebases are not intended to be merged into one.

Overall, the definition of consistency is complex and context-dependent, and one should be aware of the limitations of the chosen definition.

B. METRICS

Metrics are widely used in software engineering to assess the quality of the codebase, the development process, and the developed software [21]. The ISO/IEC 25023 [22] provides a well-established overview of quality measures and their purpose in software engineering.

Generally speaking, a metric can range from a simple code-quality metric to a complex process-success indicator. The following aims to give an overview of the different natures and applications of metrics in software engineering. Of course, this list is by no means exhaustive.

The first metrics a software developer stumbles upon are typical code-quality metrics such as lines of code, cyclomatic complexity, or coupling and coherence of object-oriented code [23], [24]. These metrics are retrieved from the codebases using static analysis, i.e., processing the source code without executing it. More complex quality indicators can be derived from such simple metrics and suitable heuristics. A number of approaches have proposed to combine simple metrics into aggregated measures or indices. For examining the maintainability of a software system, several works focus on finding a so-called *maintainability index* [25], [26], [27]. The maintainability index is a composite metric that is calculated as a weighted sum or a polynomial of selected simple metrics.

Apart from metrics derived by static analysis, information can also be gathered from executing the code, with or without added instrumentation. This is called *dynamic analysis* [28], [29]. A well-known example is the family of test-coverage metrics [30]. Environmental metrics such as runtime and memory consumption can also be included in this category.

Repository metrics assess the quality of software repositories [31]. For co-located development teams, such metrics provide valuable insights into the development process.

Research on metrics for consistency assessment focuses mainly on consistency within an artifact or between stages of the development process, such as assessing the consistency of requirements specifications [32], [33]. Another special case is the well-researched family of metrics for variability models [34], [35] or feature models [36]. An approach to measure the consistency of a software architecture description is the use of entropy, as investigated by Niepostyn and Daszczuk [37]. Their rather formal approach to the problem is based on the specification of consistency rules to prescribe the desired consistency requirements.

On the other side of the spectrum, all these kinds of metrics can be used for product and process quality improvement. Metrics play a key factor in the concept of managed evolution, as described by Furrer [38], or in the development of sustainable software architectures [39]. Often, the decision to deliver quickly or to cut expenses is prioritized over a well-designed software. The negative effects from this practice are summarized as *technical debt* [13], [14].

C. MERGING

Lastly, we give a brief overview of software merging as merge conflicts form the basis for this work's branch-drift metric. Merge conflicts are a well-known problem in software development [6], [7], [40]. They occur when the same part of the code base is changed in two different branches in different ways. Apart from the well-know textual merge algorithms, merges can also be performed on the abstract syntax tree (AST) of the code base [8]. Such advanced merge algorithms reduce the number of false positive merge conflicts, i.e., merge conflicts that are not actual conflicts but side-effects of the merge algorithm. Shen and Meng built a benchmark for merge conflicts based on empirical data [41]. They found a non-negligible number of false positive merge conflicts.

A different family of approaches tries to detect and predict conflicts using machine-learning methods based on the current code base, past development data, and supplementary artifacts such as the commit history [10], [11].

Despite this large body of literature, a metric for proactively measuring the potential of merge conflicts as an early indicator of increasing technical debt has not been presented yet.

III. THE DRIFT METRIC

This section provides a comprehensive overview of the drift metric for modeling as introduced by Kegel et al. [15].

In modeling, e.g., using UML [42], large models are edited by a team of collaborators. Model artifacts are typically stored and distributed using a model management system with features similar to Git. During project work, engineers create variants (branches) of a model that are merged at some point. Another scenario in modeling is the creation and propagation of *Views* instead of full variants [43], [44]. A view can be considered a subset of all artifacts of a full branch. Models are not compared via string-based algorithms as used by Git by default. Instead, syntactical structured differencing algorithms are used. These algorithms, such as *EMFCompare*, provide accurate, detailed conflict reports containing conflicting model elements on the syntactical level [45]. Although such approaches are also known for code [8], [9], [46], they are less established in the programming world due to the heterogeneity of programming languages and project setups. The drift metric enables tracking and visualizing the conflict potential in a set of model variants. In theory, the metric works with any kind of pairwise distance (syntactical, string-based, semantics, etc.) as long as it returns a single finite value representing the number of conflicts.

A. DRIFT CALCULATION

First, the pairwise conflict count must be computed for all pairs of existing variants to compute the drift metric for a set of model variants. The pairwise comparison algorithm is abstractly called Δ function. If M is the set of models, then $\Delta : M \times M \to \mathbb{R}$. We then use the calculated distances to build up the conflict matrix $\mathbf{A}$. The number of model variants determines the size of this matrix. Each matrix cell contains the computed distance for the respective variant pair. Third, this matrix of "conflict distances" is the input to calculate the embedding of all variants as points into the 3D space using *Multidimensional Scaling (MDS)* [47]. The result is a point-cloud visualization of all variants. The axes of the target coordinate system are linear and unit-less. The distance between two points in the point cloud is approximately proportional to the conflict count between the two variants. Variants close together indicate view conflicts, and variants far from each other indicate many conflicts. Embedding many variants into the 3D space increases stress, i.e., the embedding is imperfect. Finally, the scatteredness of this point cloud defines the value of the drift metric. The median average deviation (MAD) serves as the scatteredness measure [48]. The MAD of a dataset $\{x_1, x_2, \ldots, x_n\}$ is defined as

$$\widetilde{d}_{0,5}(x, \ldots, x_n) := \frac{1}{n} \sum_{i=1}^{n} |x_i - \tilde{x}|$$

B. EXAMPLE

Let a repository contain three branches: a, b, and c. Each pairwise branch combination may be the source for merge conflicts. To identify the conflicts present, we attempt a merge for each pair of branches and store the number of conflicts in matrix A. This leads to $3 \cdot 3 - 3 = 6$ merge attempts. If we assume a symmetric merge operation, e.g., merging a into b is the same as merging b into a, we can halve the number of merge attempts to 3. The result could be the following matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix}$$

The rows and columns are in the order of a, b, and c. The matrix is symmetric along the diagonal, and the diagonal contains only zeros as no branch has conflicts with itself. The three branches, a, b, and c, have different conflict counts with each other. The next step is to compute the 3D point cloud from the matrix A via an MDS algorithm. We find the following points $\dot{a}$, $\dot{b}$, and $\dot{c}$ using the *sklearn.manifold.MDS*[4] library in Python (rounded on second decimal):

$$\dot{a} = (0.10, 0.46, 0.03)$$
$$\dot{b} = (0.49, 0.82, -0.87)$$
$$\dot{c} = (-0.60, -1.28, 0.84)$$

Although the resulting coordinate system is unitless, the distances of the 3D points are approximately equal to the distances in matrix A. We can check this property by calculating the Euclidean distance between the points for a and b in the point cloud: $d(\dot{a}, \dot{b}) = 1.044 \approx 1$. However, for larger matrices, having a certain amount of stress, i.e., a partly suboptimal embedding, is likely. The drift value is calculated from the point cloud using the MAD. The geometric median of the point cloud is $(0.10, 0.49, -0.60)$. The average distance of the point cloud from the geometric median is: $\widetilde{d}_{0,5} = 1.19$. Consequently, the drift value of the repository is 1.19.

C. VISUAL INTERPRETATION

Figure 3 shows the schematic example of a drift trend line and a point cloud for each drift value. The point clouds are depicted in 2D for better readability. Each point represents a variant, i.e., a branch where a collaborator works on some feature. The time passes from left to right. Assume that all axes are scaled linearly and equally in all plots a - d. From the first to the last plot, the scatteredness of the point cloud increases. The drift value increases as well because it is described by the MAD of the point-cloud. The main reason for the increasing drift can be visually identified as an outlier (the red point) that moves away from the other points over time. This indicates that merging the red colored branch with any other branch becomes increasingly difficult, i.e., the technical debt increases.

[4] https://scikit-learn.org

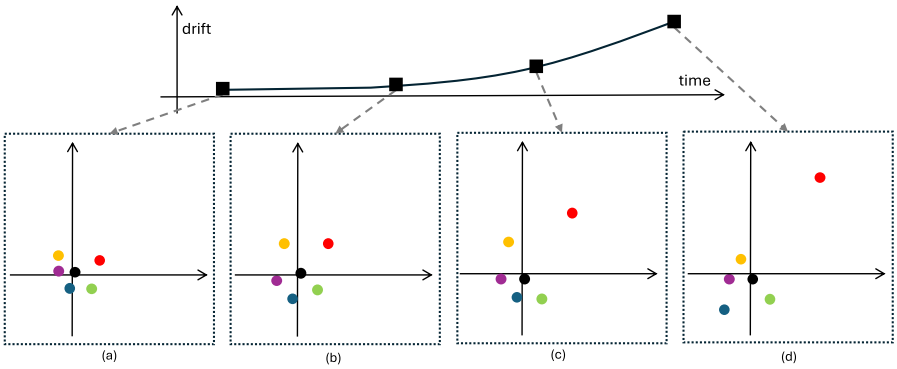

FIGURE 3. Exemplary evolution scenario of a drift increase over time together with its visual representation as a point cloud. The red outlier responsible for most of the drift increase is easy to spot.

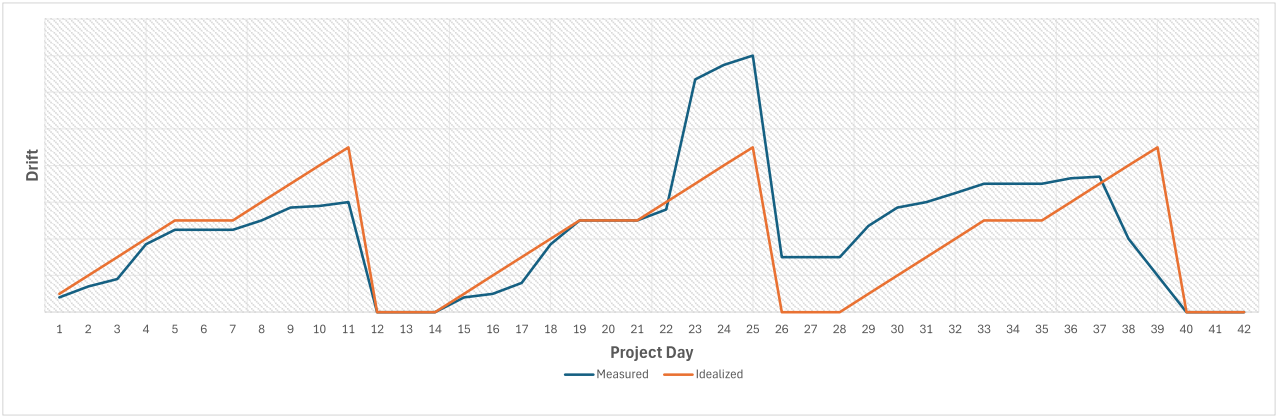

FIGURE 4. The orange plot shows an arguably good drift chart for an agile process with a two-week sprint cycle. The blue drift chart shows a potential reality with hotspots and irregularities. Both charts are synthetic examples. The Y-axis shows the drift value using a linear scale. Concrete Y-values are omitted as only the shape is relevant.

IV. DRIFT IN SOFTWARE DEVELOPMENT

This section transfers drift to branch-based software development. We further discuss aspects of drift computation that are relevant under real-world conditions. The work by Kegel et al. already proposed transferring the metric to branch-based software development [15]. However, the authors evaluated this proposition only briefly by providing a limited set of examples. The challenges resulting from the practical conditions of typical software development projects have not been addressed yet. However, looking at branch-based software development in depth, we can identify several differences to the modeling context. Henceforth, we refer to the drift metric instantiated for branch-based software engineering as *branch-drift*.

A. REQUIREMENTS FOR BRANCH-DRIFT COMPUTATION

The main requirement for computing drift is a suitable pairwise distance measure for the set of variants. Our variant set is the set of branches in a version control system. As each branch contains a (potentially modified) copy of the codebase, we need a distance measure to compare these heterogeneous artifacts. This distance measure should be as stable as possible, i.e., small changes in the code base should not lead to large changes in the calculated distance. Additionally, the distance measure should be adequately fast, as the number of comparisons grows quadratically with the number of branches. The second requirement is the "purposefulness" of the individual branches. Each branch has to represent an equally important development task. Furthermore, there should exist no branches that are either not intended to be merged or do not contain a modified copy of the original but are used for other purposes.

B. CONDITIONS IN SOFTWARE DEVELOPMENT

The following paragraphs summarize the "as is" conditions facing software development projects. We derived the following statements from our experience, informal discussions with practitioners, and the lessons from preliminary experiments from this work's evaluation.

a: EXPECTED DRIFT TREND

Figure 4 shows a schematic example of two drift chart lines in a SCRUM-like [49] agile process with a two-week sprint cycle. The orange line shows the expected case with

a growing drift value during the sprint. At the end of the sprint, all changes should be integrated into a single version. The blue line shows a potential reality with hotspots and irregularities. In the second sprint, the drift value overshoots its expected boundary - a possible sign of a problem in the development process. Consequently, the drift does not reset to zero at the end of the sprint.

b: BRANCHES

The conditions and practices in software development projects can differ significantly depending on the technology, team, and development process. This work focuses on feature-branch workflows, i.e., development processes where each feature is developed in a separate branch. In other words, each development task leads to a new branch, and merging this branch back into the main branch marks the task's completion [4]. However in practice, this process is not always followed strictly. Surveying open-source repositories for our evaluation, we found various ''misused'' branches, e.g., for documentation only, release planning, and webpage maintenance. Another issue in community-driven open-source projects is the large number of ''outdated'' branches. These have not been actively developed for a long time but are still present in the repository for various reasons. However, many of these branches could actually be ''dead,'' i.e., they no longer serve a purpose. Including these branches in the branch-drift calculation shifts the results to a higher drift value by adding a constant number of conflicts to all other branches.

c: MERGING

In contrast to the conditions in modeling projects, a branch in software development is a highly diverse collection of files. These files may have different types, formats, encodings, relevance, and purposes, making it difficult to compare branches to calculate a meaningful distance measure. Ideally, one uses structured (AST-based) differencing algorithms for each file type involved [8]. However, apart from the fact that such algorithms are only available for a limited set of formats and languages, whether the resulting conflict counts are comparable remains a question. The common diff and merge algorithms in software development are string-based and work on a line-by-line basis. Advanced variations of these algorithms also consider the line context, i.e., the surrounding block. If a merge conflict occurs, it manifests a number of mismatching lines. The merge algorithm does not provide any information about the nature of the conflict. A complex refactoring or a simple reformatting could cause a merge conflict involving multiple lines of code.

d: HETEROGENEITY

As mentioned before, typical software projects contain a heterogeneous set of files. Even under the assumption that the comparison algorithm can handle all file types adequately, there are files where the developer does not, or should not, care about conflicts. This includes, for example, log files, generated code, or build artifacts. Even if there are conflicts in these files, they are typically resolved by deleting the file, regenerating it after the merge, or overwriting it with the most recent version. If we assume the usage of a standard string-based diff algorithm, more issues arise. Conflicts in binary files, such as images or compiled code, are only detectable per file and not fine-grained. Also, the weight of a line of code can vary between file types and even developers. Some file types contain more lines than others just for syntactical reasons. For example, a *JSON* file contains more lines than a *YAML* file for the same content. Many programming languages allow for multiple ways to write and format code. For example, in *Java*, one could write a whole class in a single line or each statement on a separate line. If some developers prefer writing compact code, they will have fewer conflicting lines, although resolving conflicts in such code may be more difficult.

e: PROJECT SIZE

The size of a software project can vary significantly. The repositories investigated during this work range up to 800 branches and 6 GB in size. This means that tools for calculating branch-drift must be aware of these potential sizes and be able to handle them efficiently.

f: ANALYSIS CONDITIONS

The state-of-the-art version control system is Git, and the most common platforms for hosting Git repositories are Github and Gitlab. In recent years, continuous integration and continuous deployment (CI/CD) pipelines have become a standard on these platforms [50], [51]. This enables the execution of automated quality assurance and testing processes for each commit. Consequently, the branch-drift metric calculation must be integratable into these pipelines.

C. BRANCH-DRIFT COMPUTATION

In software development, we must cope with poorly managed non-feature branches, heterogeneous file types, string-based diff algorithms, large project sizes, and CI-specific technical requirements. This threatens the feasibility of the branch-drift metric under realistic conditions. Furthermore, technical measures are required to handle these conditions before and during the drift calculation. Kegel et al. introduced the *Driftool* for calculating the drift metric for Git repositories [15]. The tool is a prototypical implementation written in Python. The Driftool automates Git commands to attempt merges automatically on all branch combinations. The distance measure used is the number of conflicting lines during the merge. To cope with the requirements of typical Git repositories, Kegel et al. realized the following features for configuring the analysis:

- **Blacklist**. File types matching the regular expressions in this list are ignored during the merges.
- **Whitelist**. Only file types matching the regular expressions in this list are analyzed for conflicts.

- **Branch-ignore list**. Branches matching the regular expressions in this list are excluded.

These features are a good starting point for handling the above-mentioned conditions. In particular, ignoring irrelevant and problematic file types is easily possible. The possibility to ignore branches enables the explicit exclusion of non-feature branches, as long as naming conventions are followed to formulate an according ignore rule. However, we still identify features we consider necessary for meeting our requirements.

- To handle outdated branches, we propose a **timeout** feature. If a branch is inactive for a specified number of days, it is automatically excluded from the analysis.
- To receive results less biased by the different file types, we propose extending the single numeric drift value to a **drift vector of three values**. The three values are the drift values based on the (1) number of conflicting lines, (2) number of conflicts, and (3) number of conflicting files.
- To improve performance, we propose using multithreading to **parallelize the merge process**.
- To enable the integration into CI/CD pipelines, we propose the **platform-independent deployment** of the tool as a *Docker* container. We propose further runtime optimizations inside the container, e.g., the usage of a *RAMdisk* for slow repository file operations.

Realizing the proposed features leads to a complete re-implementation of the original Driftool. We give an overview of the extended tool in section V. The question of whether these measures are sufficient to handle the discussed conditions is the target of this work's evaluation in sections VI and VII.

V. THE DRIFTOOL

This section introduces the *Driftool*, a tool for computing the branch-drift of a Git repository. As already mentioned in Section IV, we re-engineered and re-implemented the former prototypical version of the Driftool from [15].

Realizing a full-fledged tool for branch-drift computation is one of the main contributions of this work. A metric without a usable tool for its computation would limit both its experimental evaluations and practical applicability. The previous version of the Driftool was implemented in *Python*. For the new version, we decided for a *Kotlin* implementation as Kotlin better supports a clean object-oriented design.

A. ARCHITECTURE AND FUNCTIONALITY

We briefly describe the architecture and functionality of the Driftool in the following. Figure 5 illustrates this process using the notation of a (rather informal) UML activity diagram.

The analysis starts with the client selecting the repository to be analyzed. If not yet present, an analysis configuration file must be created. The configuration file contains - among others- the blacklist, whitelist, and branch rules, as well as the timeout value. An example configuration file is shown in listing 1. The client then starts the Driftool by executing the provided *run* script that creates a new running instance (container) from the Driftool docker image. Having the Driftool deployed inside a docker container ensures platform independence and easy deployment, particularly within a CI pipeline. Using an in-memory volume (RAM-disk) inside the container speeds up the execution of Git commands compared to a disk with less bandwith. After the setup, the actual Driftool Kotlin application is executed within the container.

The application starts by preparing the repository based on the configuration. Hereby, the application heavily relies on the execution of Git subprocesses and the parsing and evaluation of their outputs. First, the list of branches is fetched. Irrelevant branches are filtered out based on the branch names and the branch-ignore rules. For each remaining branch, the last commit date is fetched and compared to the timeout value. Branches that were not ignored and are not outdated form the branches of interest. Now, for each branch of interest, the blacklist and whitelist rules are applied. We call the resulting fully prepared repository the *reference repository*. The application determines all merge jobs from the list of branches of interest. Depending on the configuration, symmetric pairs are included or excluded (analysis a-b and/or b-a). A copy (in-memory) of the reference repository is created for each thread. Each thread receives and executes its merge jobs. The results are joined into the distance matrix. We are interested in three distance values (lines, conflicts, files), so we handle three distance matrices. The Driftool calculates the 3D point-cloud embeddings and drift values from the distance matrices and writes them to report files. The report files, also called *drift reports*, are serialized in JSON format. A drift report furthermore contains all relevant intermediate data from the computation process, e.g., the branch list, analysis time, and distance values. The container copies the results and logs to the client's host system. The container self-destructs on completion.

```
1  reportIdentifier: Example Analysis
2  timeoutDays: 60
3  fileWhiteList:
4      - "\\.java"
5      - "\\.kt"
6  fileBlackList:
7      - "build\\/"
8      - "dist\\/"
9      - "gen\\/"
10 ignoreBranches:
11     - "^release\\-"
12     - "^v\\."
```

LISTING 1. Configuration example.

1) COMPLEXITY AND PARALLELIZATION

According to the internal design of the Driftool, we distinguish between three states of the repository under analysis:

- *Unprepared State*: The repository as it is provided by the client.

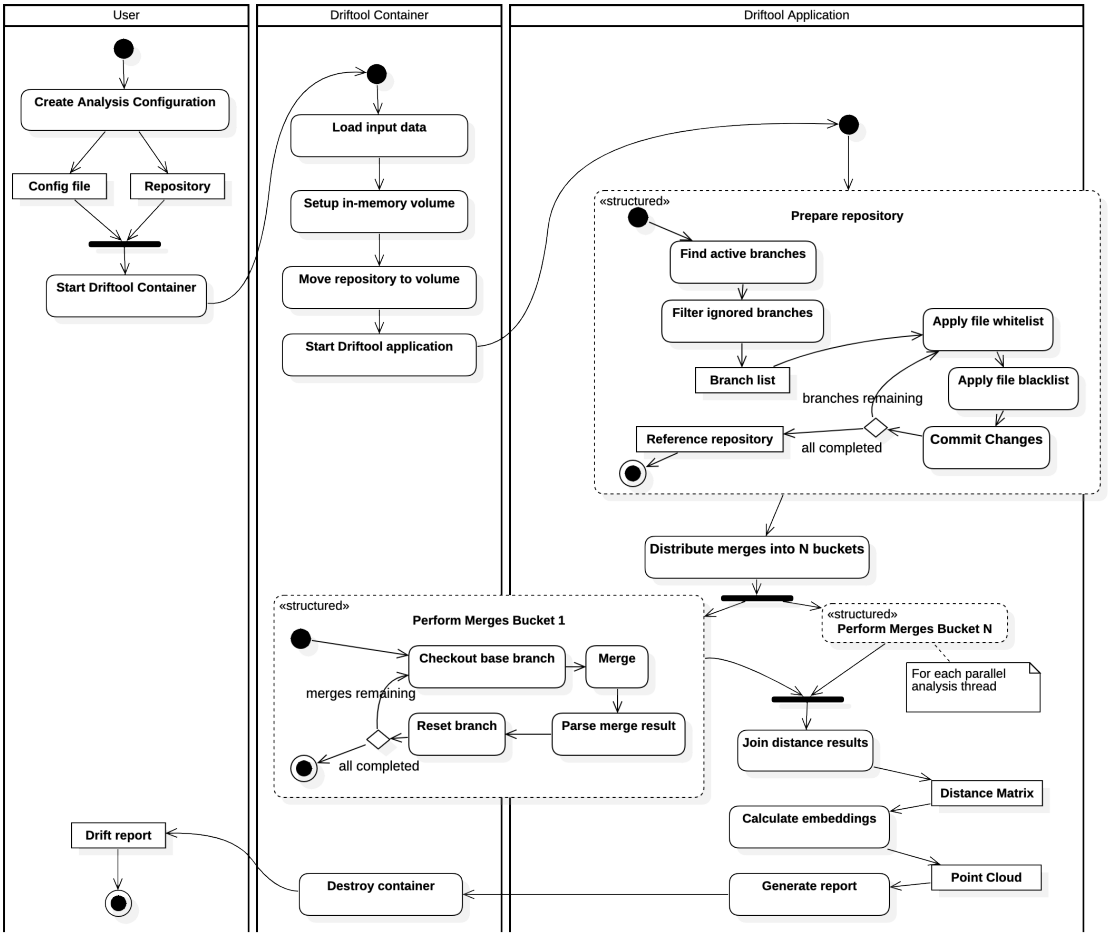

FIGURE 5. The conceptual overview of the activities involved in the drift computation process via the Driftool. The majority of the shown activities interact with the input Git repository, retrieve meta-information, and manage automated merge attempts. The actual drift calculation as introduced in Sec. III only happens in the last steps of the depicted process on the bottom right side.

- *Prepared State*: All branches of the repository are visited, and branches of interest are checked out. The analysis blacklist and whitelists were applied to all branches of interest. This is also the state of the *Reference Repository*.
- *Partially Prepared State*: The *Prepared State* is reached only for a subset of branches of interest.

A Driftool run comprises two time-intensive steps: Transitioning from an unprepared repository to a prepared repository and conducting all merge operations. Preparing a repository is time intensive as the tool must visit all files, possibly delete them, and commit the changes. Therefore, the slow *git add –all* command is required. The branch visiting has a linear complexity, as it takes a certain amount of time to visit each branch. The merge procedure is naturally computation-heavy. The potentially large number of merges, parsing the merge results, and resetting the repository state after each merge requires a relevant amount of time. The number of merges grows quadratically with the number of branches. Assuming a symmetric merge operation, the number of merges required to construct the distance matrix is $(n^2 - n)/2$. The full merge procedure requires a prepared repository, whereas a single merge operation only requires a partially prepared repository.

The central optimization approach is the parallelization of the drift computation. Both the repository preparation and the merging can be parallelized. We consider parallelizing the merge operations more relevant due to the quadratically growing number of merges. We realized this optimization as already mentioned in the previous Section.

Parallelizing the repository preparation is more complex as it is not safe to use multi-threaded Git commands on a single repository. The required file system operations would conflict. However, one could partition the merge jobs in a way that each thread works with a small subset of branches only. Instead of a prepared repository, the unprepared repository is passed to the threads. This way, each thread could create its own partially prepared repository in parallel. Although we considered this optimization step, we did not implement it in the current version of the driftool as this would significantly increase the difficulty of debugging and monitoring.

Apart from the parallelization, one technical bottleneck of the drift computation is the disk I/O (considering a common setup with a fast CPU paired with a slow disk). The executed Git commands are not particularly CPU-demanding but perform many file system operations. Therefore, we designed the execution environment inside a VM (docker) to create the internal repository copies on a RAM-disk, i.e., in the main memory of the docker container.

B. TECHNICAL LIMITATIONS

We extended the Driftool with all the functionality required to use it on any Git project. The tool contains a comprehensive testsuite for verifying its internal functionality, such as Git automation, drift computation and report generation. However, as the tool heavily relies on the automated execution of Git commands, not all edge cases are covered. During the evaluation, we encountered rare cases in which the tool did not work as intended. Underlying errors were edge cases related to system settings, encodings, or Git. We give two examples of such limitations in the following: (1) One repository we tested contained branch names that Git was not able to checkout without further trickery as they were named numerically like commit hashes. Although we found a workaround for this problem, it is very specifc and we decided not to include it in the tool. (2) Some repositories we tested caused errors in our automation process due to the encoding of file names and the usage of (potentially broken) symbolic links in the repository. As we consider this a highly repository- and platform-specific problem, we decided not to investigate and resolve such issues further. Files that are not UTF encoded, such as binary blobs, are handled by the tool safely. (3) One repository we tested contained branches without a common base with the main branch. We implemented a policy of skipping such branches, as merging would require additional parameters and may lead to unexpected results.

We implemented throughout exception handling and logging. In case the tool encounters an error, the drift calculation terminates in a controlled fashion, and the user receives all relevant information.

VI. STUDY DESIGN AND SETUP

This section presents the design of the experimental evaluation. We conducted a study on real-world software repositories in which we calculated and analyzed the branch-drift metric in several experiments. This section includes the presentation of the evaluation objectives, procedure, and setup. The results are presented in the following section VII.

A. OBJECTIVE

This work investigates whether branch-drift is a suitable and efficiently computable metric for inconsistency monitoring in feature-oriented software development. We do not aim to show general applicability of the metric. The objective is to show the suitability and efficient computing for repositories meeting the target scope of the branch-drift metric. We also do not aim to show the correctness of the branch-drift metric, i.e. that it calculates a measure of inconsistency. This property directly results from the metric's definition and is not subject to evaluation. We aim to show that branch-drift is useful for the intended purpose. We accept branch-drift as *suitable* (RQ1/RQ1.1) if it fullfills the following three hypotheses:

- **RQ1.H1** Branch-drift does not show a strong correlation with an activity metric.
- **RQ1.H2** Branch-drift shows hotspots (high increase or decrease in a short period) that are explainable by correlated hotspots in the repository's activity metrics.
- **RQ1.H3** Branch-drift shows hotspots that are not explainable using the repository's activity metrics.

Accepting RQ1.H2 shows the existence of scenarios where branch-drift is explainable by intuition, e.g., a peak in activity increases inconsistency. Accepting RQ1.H3 shows that branch-drift leads to an actual information gain that could not be achieved by only looking at the repository's activity metrics. We evaluate the three different branch-drift variations with different distance measures, namely conflicting lines, number of conflicts, and conflicting files. For accepting the hypothesis a single fulfilling branch-drift variation suffices. To answer RQ1.2, we do not state a hypothesis but investigate and interpret the branch-drift data resulting from the suite of experiments.

We accept branch-drift as efficiently computable (RQ2) if the computation times lie within the magnitude of less than 1 hour for large repositories. However, we consider a computation time of less than 1 minute ideal. The calculations must happen on a consumer-grade workstation machine, i.e., without any high-performance computing infrastructure. We state the following hypothesis for RQ2. The acceptance of hypothesis RQ2.H1+ is considered the ideal case but is not mandatory for the acceptance of RQ2.

- **RQ2.H1** No branch-drift computation (single run) out of all investigated repositories takes over one hour.
- **RQ2.H1+** No branch-drift computation (single run) out of all investigated repositories takes over one minute.

B. PROCEDURE

The study's procedure is divided into 6 main steps. These steps do not include the development of tools and methods, which was conducted prior to the study.

1) Compile a list of repositories to perform the experiments on. The repository selection process is described in Sec. VI-C.
2) Clone the selected repositories every working day over a fixed analysis period. We present an overview of the selected repositories in Sec. VI-D.
3) Create an analysis configuration for each repository. The configuration details are stated in Section VI-E.
4) Process each clone of each repository using the Driftool. This produces a drift report for each clone.

5) Fetch the activity metrics of the selected repositories from the Github API. Section VI-F provides details on the activity metrics.
6) Analyze the drift reports and activity metrics. This includes generating visualizations and comparing the branch-drift data with the activity metrics.

To investigate RQ1.H1-H3, we compare the derived branch-drift reports with available repository statistics. These statistics include commit activity, release activity, and number of active branches. To evaluate RQ2.H1 and RQ2.H1+, we analyze the runtime of the Driftool with varying configurations. We provide insights into our technical setup and operation configurations in Section VI-H. The specifications and purposes of the different derived analysis runs are stated in Section VI-G.

C. REPOSITORY SELECTION

For this work's experimental evaluation, we chose to analyze a small set of repositories, ranging in number from 20 to 30. We consider this number sufficient for evaluating the feasibility of the branch-drift metric but not too large to still allow for manual inspection of the individual results.

As we stated in previous sections, we designed the branch-drift metric for feature-oriented software development. Therefore, we need to identify repositories that are maintained using such a workflow. We refer to this requirement as the *Workflow* criterion. To justify this decision, we recall the definition of the branch-drift metric. Branch-drift is defined based on the condition that branches exist to be merged into the main branch at some point. For example, suppose a repository contains only permanently existing branches, such as individual software components that are not intended to be merged. In that case, it makes no sense to calculate a measure of inconsistency based on merge conflicts. In this exemplary case, other analysis techniques, such as static analysis to check for inconsistencies based on not matching interfaces, would be more suitable. Naturally, we do not expect open-source repositories with community participation to follow a particular workflow perfectly. However, we are confident that the metric and its implementation are sufficiently robust to cope with deviations from a clean feature-oriented workflow as long as its basic principles are implemented.

Apart from the *Workflow* criterion, we furthermore defined the following criteria for our selection process:

1) *Availability*: Publicly available on Github; Licensed under an open-source license.
2) *Activity*: Under active development; Multiple branches with recent commits.
3) *Size*: Sufficient number of contributors, commits, and branches

We decided on the additional criteria of *Size* and *Activity* as we do not expect any non-trivial insights from, for example, analyzing a small repository with only two branches and a single contributor.

However, deciding whether a repository is suitable for our analysis based on the above criteria is no simple threshold-based selection. Instead, we conducted a fully manual selection process. We browsed Github for repositories that are well-known and widely used in the open-source community based on our own experiences. To limit the selection process, we focused on repositories hosted by well-known organizations or companies, such as *Meta* (Facebook) and *Microsoft*. We assume that these organizations follow well-defined workflows and manage their repositories professionally, i.e., with branch naming conventions, issue management, and contribution guidelines in place. Each candidate repository was assessed by the research team based on the above criteria. The final selection was made by consensus. The result was a set of 30 candidate repositories.

We later excluded nine repositories from the dataset during the cloning process. The exclusion reasons were either technical: The Driftool was unable to reliably analyze the repository in test runs (see V-B). In other cases, the exclusion was the consequence of a miss-selection: The branch-drift was zero throughout the analysis interval due to a lack of activity; The purposes of the repository's branches turned out to be not clearly identifiable from their naming schema during the configuration phase. The exclusion was based only on the repository data and plain Driftool results. The exclusion happened before the in-depth data analysis and comparison.

D. REPOSITORY DATA ACQUISITION

After the selection process, we manually cloned the selected repositories every working day for three months. Choosing a concrete period is always a trade-off between the amount of data and the time needed for the analysis. We consider three months sufficient to observe full feature-branch lifecycles (fork, develop, merge, delete) while still keeping the analysis time and data size reasonable.

We started the cloning process on the 26th Feb 2024 and ended it on the 31st of Mai 2024. For efficiency, we excluded weekends from the cloning process. Preliminary experiments showed a highly reduced activity on weekends, which is also supported by the activity metrics presented in section VI-F. Cloning each repository on each working day is necessary as cloning a repository only once and then analyzing its internal history can result in an incomplete history. Branches created but later deleted, or changes that may have been undone by reverts or *force* operations would be missing.

The final dataset comprises 21 repositories cloned five times a week over 14 weeks. This results in 1470 repository clones totaling over 1.2 TB of data. Table 1 shows the overview of the final repository dataset. We already provide the repositories' total number of branches and number of analyzed branches (potential feature branches, commit within 60 days) on the acquisition's starting day. We also provide the identifier of each repository's open-source license, a short description, and the primary programming language(s).

The table shows that we classify 11/21 repositories as frameworks, only 4/21 as applications, 4/21 as platforms or

TABLE 1. Cloned repositories in the final analysis dataset. The table shows the *Github* organization, the repository name, type, the total number of branches on the first day of the analysis, the number of analyzed (relevant) branches on the first day of the analysis, license, description, and the primary programming language(s). The organization- and repository names are denoted as in their *Github* URL. The type field distinguishes A: application, F: framework, P: platform/language.

ID	Organization	Repository	Type	Branches Total	Branches Analysed	Open-Source	Short Description & Main Language(s)
M1	microsoft	datashaper	F	23	10	Yes (MIT)	A framework for data processing with focus on web development. (TypeScript, Python)
M2	microsoft	PowerToys	P	77	32	Yes (MIT)	A set of Windows system utilities, e.g., *Color Picker*, *Command Paletter*, or *Mouse Utilities*. (C#, C++)
M3	microsoft	semantic-kernel	F	65	13	Yes (MIT)	A framework/SDK for building AI agents with LLMs. (C#, Python)
M4	microsoft	terminal	A	503	36	Yes (MIT)	The Windows Terminal and console host implementation. (C++)
M5	microsoft	TypeScript	P	297	12	Yes (Apache-2.0)	Implementation of the TypeScript web programming language. (TypeScript)
M6	microsoft	vscode	A	852	79	Yes (MIT)	The Visual Studio Code (VSCode) IDE. (TypeScript)
F1	facebook	lexical	A/F	136	11	Yes (MIT)	An extensible text-editor framework build with web-technology. Used for web-based code editing. (TypeScript, JavaScript)
F2	facebook	Rapid	A	50	6	Yes (ISC)	An web-based editor for map editing on *OpenStreetMap*. (JavaScript)
F3	facebook	react-native	F	212	32	Yes (MIT)	A framework for developing react-based applications for technical domains outside the web. (C++, JavaScript, others)
F4	facebook	react	F	84	13	Yes (MIT)	A large-scale library for building websites and web-based user interfaces. (JavaScript, TypeScript)
F5	facebook	docusaurus	A/F	36	7	Yes (MIT, CC)	An application framework for quickly building and deploying project websites, e.g., project documentation. (TypeScript)
K1	Kotlin	dokka	A	350	21	Yes (Apache-2.0)	An API documentation engine for Kotlin that generates documentation sites from code comments. (Kotlin)
K2	Kotlin	kotlinx-datetime	F	23	5	Yes (Apache-2.0)	A Kotlin library for date and time representation and processing. (Kotlin)
K3	Kotlin	kotlinx.coroutines	F	205	23	Yes (Apache-2.0)	A Kotlin library for parallel computing based on co-routines. (Kotlin)
K4	Kotlin	kotlinx.serialization	F	181	12	Yes (Apache-2.0)	A Kotlin library for object serialization to formats like JSON or Protobuf. (Kotlin)
K5	Kotlin	dataframe	F	100	10	Yes (Apache-2.0)	A Kotlin library for data-model creation and structured data processing. (Kotlin)
X1	electron	electron	P	148	34	Yes	A framework and runtime for cross-platform desktop apps based on web technologies. (C++, TypeScript)
X2	JuliaLang	julia	P	827	77	Yes (MIT)	The Julia programming language's compiler and standard library. (Julia, C)
X3	keras-team	keras	F	23	4	Yes (Apache-2)	A framework for deep-learning that different network models and backends. (Python)
X4	angular	angular	F	69	7	Yes (MIT)	A UI framework for building websites and web apps. (TypeScript)
X5	twbs	bootstrap	F	93	5	Yes (MIT)	A HTML, CSS, and JavaScript framework for building and styling websites. (JavaScript, CSS, others)

languages (large repositories with various components), and 2/21 as both application and framework.

E. REPOSITORY CONFIGURATION

We created an individual Driftool configuration file for each repository that is being analyzed. The individual configurations diverge in the values for ignored branches (RegExp) and whitelisted file types (RegExp). We configured each repository's branch-ignore values to ignore release branches, obviously bot-created branches, and obviously no-feature branches (documentation, website). Release branches exist to store a particular release of the repository separately. They are not intended to be merged. We configured a file whitelist for each repository for full control over the analyzed files. In practice, we expect an experienced developer to create the Driftool configuration only to analyze relevant and meaningful files for consistency analysis. As we are no experts nor contributors to the analyzed repositories, we cannot create fine-grained black- or whitelists. To account for that, we manually created a file whitelist for each repository based on its most relevant code file types. To do so, we reviewed the repositories' language statistics provided by Github. We aimed to include more than 90% of the repository's code files in the analysis. To compare and evaluate this choice, we conduct a second analysis run

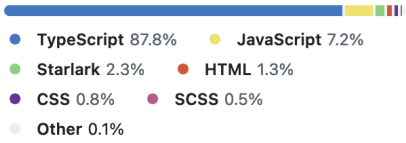

FIGURE 6. Example of a repository's language distribution statistic provided by *Github*, here from angular/angular.

without any file configurations. Figure 6 shows an example of a repository's language statistic provided by Github.

F. ACTIVITY METRICS

We use three *activity metrics* to analyze the feasibility of the branch-drift metric as defined by RQ1: Commit activity,[5] release activity,[6] and analyzed active branches. We fetch commits and releases from the public GitHub API. The number of active branches is a subsidiary result of the Driftool analysis.

1) COMMIT ACTIVITY

The commit activity of a repository shows the number of commits made per day. Therefore, the commit activity is a well-suited metric for measuring the development activity. Github counts a commit as soon as it is part of the default branch's history. Merge commits are excluded. We fetch the commit activity from the API > 6 months after the end of the analysis period. We argue that most changes made within the analysis period are merged by then. We fetch the commit activity for the full period +30 days before and after to observe potential trends better.

2) RELEASE ACTIVITY

The release activity of a repository shows the number and type of releases made per day. A release is a particular version (commit) of the repository that is marked (tagged). The release statistic is interesting because working towards a new release involves the integration of many features and defines a milestone in the development process. Releases are not a Git feature but are provided by collaboration platforms. To observe potential trends better, we fetch the release activity for the full period +30 days before and after. We do not differentiate release types, e.g., regular releases or pre-releases.

3) ANALYZED ACTIVE BRANCHES

We consider a branch *active* if it received a commit within the last 60 days. However, the term *active* is not standardized. In large open-source repositories, most branches are inactive for various reasons outside the scope of this work. We only consider active branches in our analysis. Out of these, we further ignore non-feature branches. The result is the set of analyzed active branches. Many active branches indicate a high development activity with many parallel tasks. Table 1 states the numbers for the first day of the analysis.

[5]https://docs.github.com/en/rest/metrics/statistics
[6]https://docs.github.com/en/rest/releases/releases

G. EXPERIMENT EXECUTION

We conduct a series of Driftool analysis runs, i.e., experiments, on the gathered repository clones to evaluate the stated hypotheses. The experiments are designed to gather multiple series of branch-drift data. We present the results in Sections VII-B and VII-C. Table 2 shows the specification of the experiment runs. We divide the runs into three groups: Group G1 aims to gather data to evaluate RQ1.H1 and RQ1.H2. Groups G2 and G3 aim to gather data for the evaluation of RQ2.H1 and RQ2.H1+.

In G1, we analyze the dataset over the full period of three months. G1.E2 and G1.E4 are verification runs with a varied thread count for G1.E1 and G1.E3. Any parallelization-related effect is expected to be visible in the verification runs. We expect these runs to deliver identical results, except for runtime.

In G2 and G3, we analyze the first week of the dataset to gather information about the Driftool's runtime. We vary the number of threads, the whitelist, and whether the merges are executed in both directions (symmetric) or symmetry is assumed (asymmetric). As denoted in Table 2, some runs of G2 and G3 are subsets of G1 runs. We execute all runs independently. Any overlaps that must yield identical drift results are compared for verification purposes. In particular, we use the results of the symmetric runs G2.E3 and G3.E3 to compare results with their asymmetric counterparts. As Git merge conflicts are supposed to be not dependent on the merge direction, we expect identical results for symmetric and asymmetric runs.

H. TECHNICAL SETUP

The overall setup comprises the Driftool analysis, the fetching of the activity metrics, and the analysis of the results.

1) DRIFTOOL ANALYSIS

The Driftool was executed on a consumer-grade workstation machine. The desktop computer was equipped with an Intel i7-13700K, 128 GiB of DDR5 RAM, and 2TB NVME M.2 storage. The operating system was Linux Ubuntu 22.04 LTS. During the experiments, the machine was only used for the analysis runs and supervision of the analysis (system monitor, file manager, etc.). The processor has 24 logical CPUs and 16 physical cores. Eight physical cores are so-called "performance cores" used for 16 threads. The machine showed no signs of performance degradation, overheating, or other technical issues during the analysis runs.

The repository dataset was stored locally on the machine. We used the Driftool's docker build setup. The analysis runs happened within the Driftool docker container. The docker container destroyed itself after each analysis run to ensure a clean environment for each run. The Driftool docker image uses Git in its default configuration, i.e., without modifying

TABLE 2. Experiment runs using the Driftool for gathering branch-drift data. Each run is identified by a *Group* and *ID*. Each run analysis all repositories in the dataset over an defined time interval.

Group	ID	Interval	Configuration	Purpose	Overlap
G1	**E1**	26.02-30.05 (full)	16 parallel, asymmetric, whitelist	Data for hotspot analysis	
G1	E2	26.02-30.05 (full)	12 parallel, asymmetric, whitelist	Verification G1.E1	
G1	**E3**	26.02-30.05 (full)	16 parallel, asymmetric, no whitelist	Data for hotspot analysis	
G1	E4	26.02-30.05 (full)	12 parallel, asymmetric, no whitelist	Verification G1.E3	
G2	**E1**	26.02-01.03 (w1)	08 parallel, asymmetric, whitelist	Runtime analysis	
G2	**E2**	26.02-01.03 (w1)	16 parallel, asymmetric, whitelist	Runtime analysis	G1.E1
G2	**E3**	26.02-01.03 (w1)	16 parallel, symmetric, whitelist	Runtime analysis, Verification	
G3	**E1**	26.02-01.03 (w1)	08 parallel, asymmetric, no whitelist	Runtime analysis	
G3	**E2**	26.02-01.03 (w1)	16 parallel, asymmetric, no whitelist	Runtime analysis	G1.E3
G3	**E3**	26.02-01.03 (w1)	16 parallel, symmetric, no whitelist	Runtime analysis, Verification	

the diff and merge algorithms. Analyzing a single repository for a single day happened as follows:

1) Start the Driftool docker container with all required arguments (number of threads, configuration location, repository copy, etc.). All configurations were previously prepared and are locally available.
2) The Driftool analysis runs.
3) The results (analysis report and log) are transferred to the host system.
4) The Driftool container stops and destroys itself.

We implemented a script that automated the execution for the whole set of repositories over a configurable period. To summarize the results, we implemented scripts that processed the analysis reports and wrote the relevant values into CSV files, which we later used for statistics and visualization. The supplementary material includes the used version of the Driftool, all automation scripts, and reports.

2) ACTIVITY METRICS

We fetched the commit-activity and release-activity metrics using the Github API. No special authentication or payment was required as all information was publicly available. We authenticated against the API to reduce rate limiting. We implemented scripts that called the respective Github API endpoints. In cases where the API uses pagination, several calls were made until the desired period was covered. We summarized the fetched data in CSV files. The used scripts, the fetched raw data, and processed CSV data are part of the supplementary material.

3) ANALYSIS METHOD

We analyze the gathered data (activity metrics and branch-drift reports) quantitatively and discuss the data based on created visualizations. To evaluate RQ1.H1-H3, we visually compare the branch-drift data with the support metrics. Therefore, we create plots comprising all data series on a normalized scale. Normalizing branch-drift data is valid as the metric is a relative measure, i.e., tendencies are more important than absolute values. We focus the evaluation of RQ1.H1-H3 on the run of the experiment G1.E1. We argue that the analysis with a whitelist (code files only) has the least noise. We compare the results with G1.E2 (without whitelist) to back this claim. We analyze the runtime of the Driftool analysis runs for evaluating RQ2.H1 and RQ2.H1+. We compare the performance quantitatively to investigate the impact of different configuration parameters.

VII. STUDY RESULTS

This section presents and discusses the results of the experiments. First, we present an overview of the fetched activity metrics. Second, we show and discuss the results of the branch-drift analysis. Third, we present and assess the runtime results. We can only present a selection of the results due to the number of experiments. The full result dataset is available within the supplementary material.

A. ACTIVITY METRICS

Commit- and release activities were fetched from January 26, 2024 until June 31, 2024. Figure 7 presents the overview of the commit activity of all repositories. The figure shows repositories with higher commit activity than others, which is expected due to the varying sizes of the repositories. The figure also shows a weekly pattern with reduced activity on weekends. This backs our initial assumption that the effect of our analysis setup, not assessing the branch-drift on weekends, is small. The long-term trend shows slightly lower activity at the end of February and at the beginning of May and June. These intervals coincide with typical holiday seasons. Figure 8 shows the summed-up release activity from all repositories. The chart also shows a weekly pattern with reduced activity on weekends but not as clear as for the commit activity. Overall, we see that releases happen throughout the investigated period. All repositories apart from one have at least one release in the fetched period. We conclude that the commit- and release activity metrics are suitable for our analysis.

B. DRIFT BEHAVIOR ANALYSIS

This section presents the results of the branch-drift analysis. In the remainder of the evaluation, we use the results of the experiment runs with the whitelist in place. We discuss the influence of the whitelist in Sec VII-B6.

1) DRIFT CHARTS

We created *Drift Charts* for all analyzed repositories to present the results visually. A drift chart shows the branch-drift values in their three variations (lines,

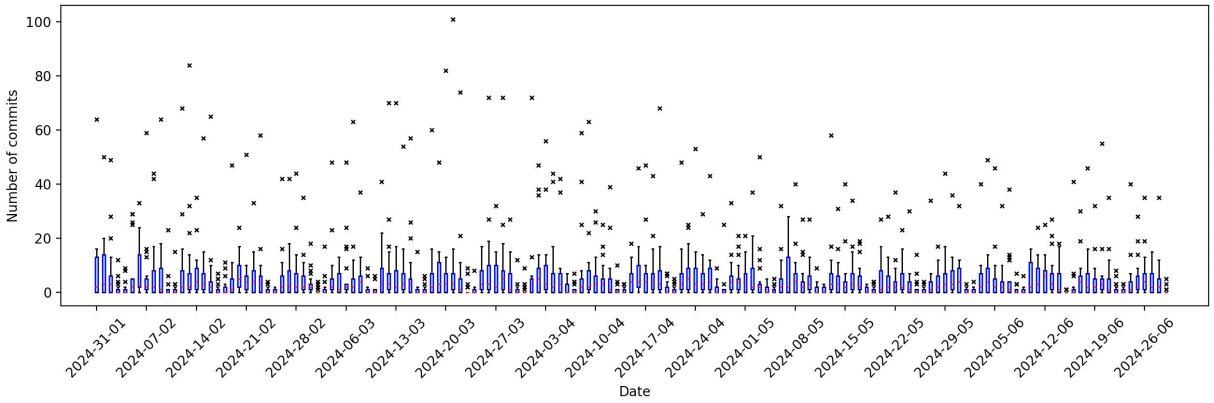

FIGURE 7. Overview of the combined commit activity from all evaluated repositories. The y-axis shows the number of commits per day.The x-axis shows the date (extended investigated interval). The chart shows the reduced commit activity on weekends.The chart shows a slight activity decrease end of February and end of Mai / June. This coincides with typical holiday seasons.

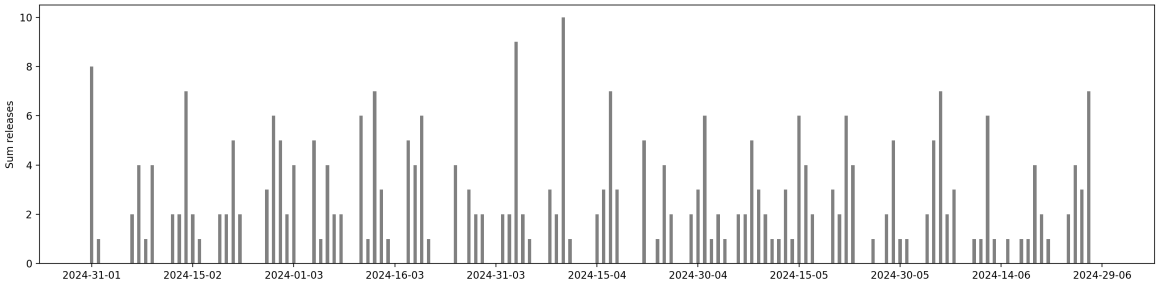

FIGURE 8. Overview of the combined release activity from all evaluated repositories. The y-axis shows the number of releases per day. The x-axis shows the date (extended investigated interval). All except one repository have at least one release in the interval. The chart shows the reduced release activity on weekends. Apart from that, no clusters or patterns are directly visible.

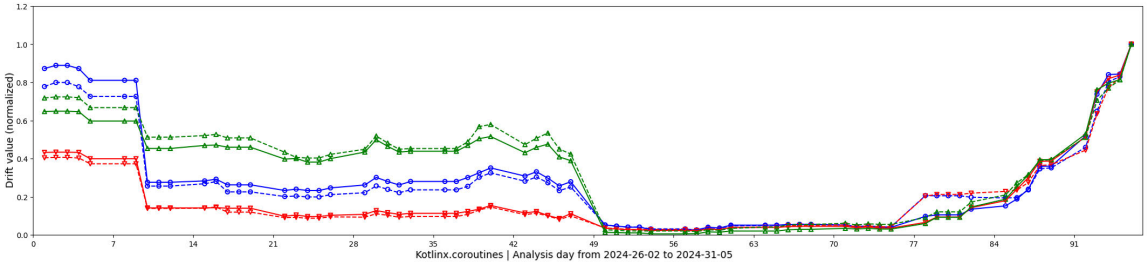

FIGURE 9. Drift comparison (with and without whitelist) for the *kotlinx.coroutines* repository. The lines show the three branch-drift variations (line, conflict, file) normalized over time. The dashed lines how the values without the whitelist configuration.

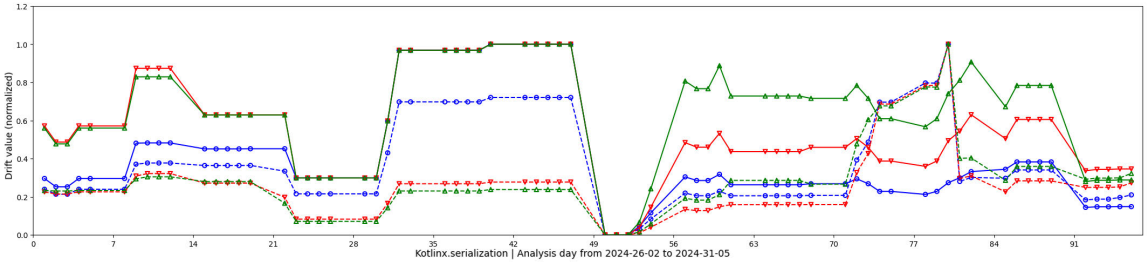

FIGURE 10. Drift comparison (with and without whitelist) *kotlinx.serialization* repository. The lines show the three branch-drift variations (line, conflict, file) normalized over time. The dashed lines how the values without the whitelist configuration. The branch-drift values without whitelistconfiguration show an additional hotspot around day 77.

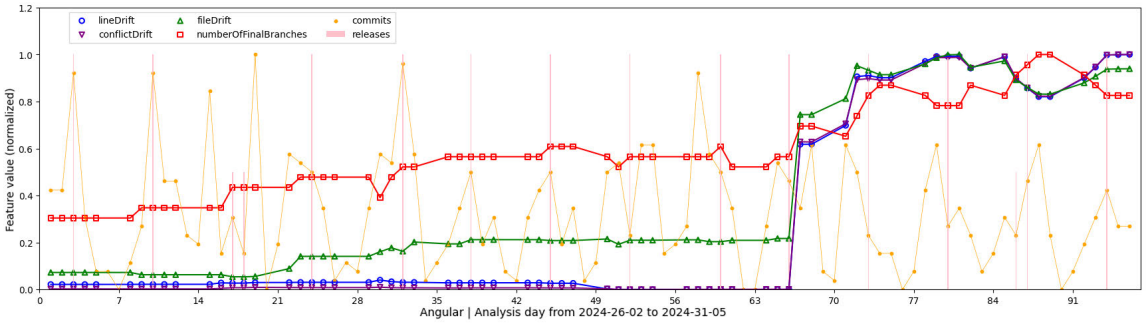

FIGURE 11. Drift and activity metrics for the *angular* repository.

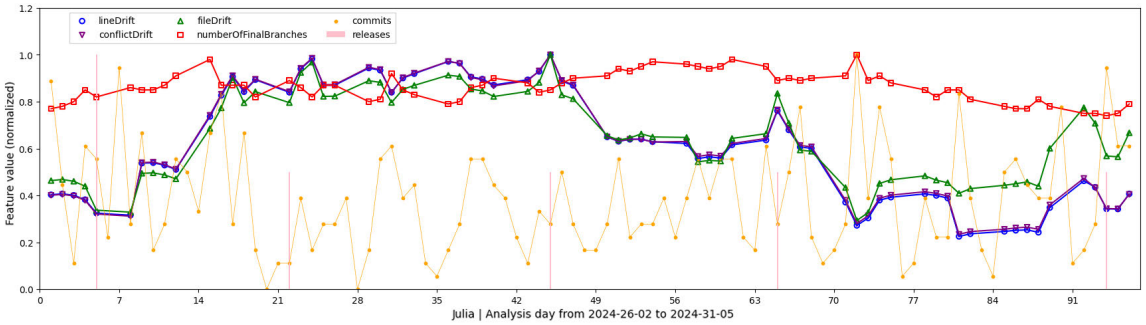

FIGURE 12. Drift and activity metrics for the *julia* repository.

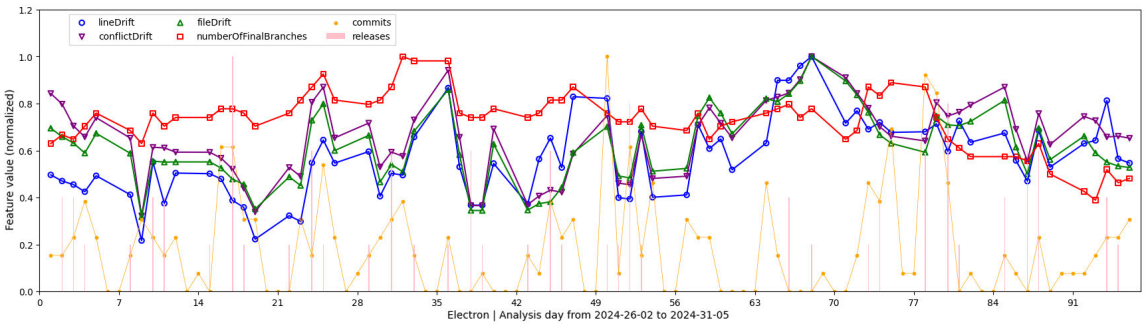

FIGURE 13. Drift and activity metrics for the *electron* repository.

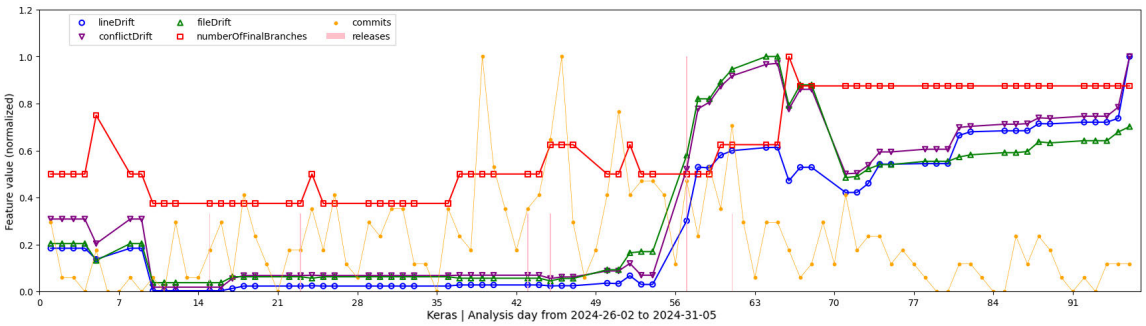

FIGURE 14. Drift and activity metrics for the *keras* repository.

conflicts, files) over time. Additionally, each chart shows the corresponding statistics (commit activity, release activity, analyzed active branches). Exemplary drift charts are presented in Figures 11, 12, 13, and 14. All values in the drift

charts are normalized from [0, max] to the interval [0, 1]. The x-axis shows the day of the investigation interval starting from January, 26, 2024 and ending on May 31, 2024. The y-axis shows the normalized values of the analyzed features. The branch-drift values are blue, green, and purple lines. The analyzed active branches are shown in red. The commit activity is shown in orange. The release activity is shown as pink bars. Markers on the chart lines indicate concrete data points. The lines are linear point-to-point interpolations between the data points. The larger gaps between data points are weekends where no data was captured.

2) GENERAL FINDINGS

This section presents general findings from the drift charts without making direct connections to our research questions. As stated in the previous section, we created a drift chart for each of the 21 analyzed repositories. Although all drift charts naturally differ, we observe common behaviors and patterns in the charts. We found that, over the observed three-month period, three stereotypical drift chart types can be identified which we present in the following. Naturally, not all repositories perfectly fit into these stereotypes.

Figure 11 shows the drift chart for the *angular* repository. The chart shows close to constant drift values in the first half of the period. The reason the file-wise drift values are at a higher level is likely an effect of then normalization; i.e., the later drift increase occurs more on the level of lines and conflicts. At around day 65, we observe a hotspot where the drift values increase drastically. We conclude that around this point, several new inconsistencies were introduced. The drift values remain high for the remainder of the period. The number of active branches increases continuously. Generally speaking, we see a stereotypical pattern of a stable drift trend with at least one clearly distinguishable hotspot that either increases or decreases the drift. We observe similar patterns for the 8/21 repositories M1, M3, F4, K1, K2, X3, X5 and X4.

Figure 12 shows the drift chart for the *julia* repository. The chart shows continuously but slowly changing drift values over the whole period. In comparison to Fig. 11, we see no clear hotspots. Generally speaking, we see a stereotypical pattern of a drift trend characterized by active upward and downward movements over several weeks. However, we see no short-term hotspots. We observe similar patterns for the 7/21 repositories M2, M4, F1, F5, K3, K5, and X2.

Figure 13 shows the drift chart for the *electron* repository. The chart shows a continuously changing drift trend. The movements oscillate around a similar level throughout the entire period. Such a pattern indicates constant activity where inconsistencies are introduced and resolved continuously. Such a pattern may indicate that the drift for such repositories should be monitored for a longer period to identify potential long-term trends. We observe similar patterns for the 3/21 repositories M6, F3, and X1. Notably, the presented electron repository shows the strongest oscillations within this group.

Three repositories (M5, F2, K4) exhibited trends that we do not confidently attribute to any of the three stereotypes. The drift charts of these repositories show behavior alternating between the different stereotypes. However, they also show no erratic jumping or extreme noise.

3) LONG-TERM BEHAVIOR AND STABILITY

The drift charts display different branch-drift behaviors for each analyzed repository. Overall, we consider the drift trend lines as well-behaved for all repositories. In other words, most repositories exhibit slowly changing drift values over subsequent days or even weeks despite ongoing development activity. Hotspots, i.e., intervals with significant changes in a short amount of time, exist but typically happen only one or two times within the analyzed period. Some repositories exhibit constant movement in the branch-drift values, but we observe no high-frequent jumping of values; rather, we see evenly distributed fluctuations on a weekly level. Considering the three-month analysis interval, this provides evidence that the branch-drift metric is suitable for long-term monitoring of repositories. If the metric had been fluctuating on a daily basis for most repositories, it would not be suitable for its purpose, as either the drift calculation, the problem domain, or the technical setup would be too sensitive to noise.

4) DRIFT VARIATIONS

The provided drift charts shows that all three branch-drift variations show similar behavior. Particularly, the drifts using line-wise and conflict-wise distances align closely. This is expected as we look at normalized values. For a large number of conflicts, the line distance is a multiple of the conflict distance by the average conflict size. However, in some cases (e.g. M5, K1, F2), the line-wise drift shows extreme values (overreactions) not present in the other Drift variations. Single large conflicts might cause these overreactions. The file-wise drift shows trends that do not align with the other drift variations on multiple occasions as in Fig. 11. This is because only the file-wise drift can count delete/edit conflicts. In conclusion, the file-wise drift is the most suitable variant for branch-drift computation if only a single value is desired. In case of a more detailed analysis, we recommend using all three variations. This conclusion answers our research question RQ1.2.

5) CORRELATING HOTSPOTS/EXPLAINABILITY

Evaluating the drift charts, we found hotspots that correlate with the activity metrics and hotspots that do not. Overall, we find no visible evidence for a general correlation between the branch-drift values and the activity metrics. We accept R1.H1 in consequence. An example of an unexplainable hotspot shown in Figure 11. Despite the drift increase around day 65, we observe no relevant changes of the activity metrics. An example hotspot that the activity metrics can explain is shown in Fig. 14. In this repository, the branch drift values increase around day 55, coinciding with a phase of increased commit activity, branching activity, and a

subsequent release. Afterward, the branch-drift values remain high while the activity metrics decrease. Similar patterns showing coinciding and non-coinciding hotspots can be found throughout the whole set of drift charts. Interestingly, most repositories show only one or two major events over the investigation interval. In conclusion, the branch-drift metric provides additional information to the activity metrics. Consequently, we accept hypotheses R1.H2 and R1.H3. This answers our research question RQ1.1.

6) WHITELIST INFLUENCE

We initially assumed that the whitelisting of code-only files reduces noise in the branch-drift metric. Comparing the results of experiment runs G1.E1 and G1.E3, we found only small differences in the behavior of the values. While values differ numerically, the normalized trends generally show similar behavior for all repositories. However, the values with the whitelist configuration show fewer hotspots for some repositories. This is within expectations as more files are analyzed in these cases. Figure 9 and 10 compare the branch-drift values for the *kotlinx.coroutines* and *kotlinx.serialization* repositories. Figure 9 shows very similar behaviour for both configurations. Figure 10 shows a case with an additional hotspot for the configuration without the whitelist around day 77. The other repositories show similar trends without or with small deviations. We discuss the influence of the whitelist on the runtime in section VII-C.

C. DRIFT RUNTIME ANALYSIS

The analysis runtime was measured for the first five days of the investigation interval from Feb 26, 2024, until Mar 01, 2024. The runtime comprises setup and analysis intervals separately. The setup interval includes preparing the reference repositories (see Section V). The analysis interval includes all steps from creating the merge jobs until the final branch-drift values are reported. Table 3 shows the average runtime per run for the repositories in the experiment groups G2 and G3. We evaluate R1.H1 directly by looking at the runtime results. No analysis run took longer than one hour. Consequently, we accept the hypothesis R1.H1. The repository with the longest runtime took less than 10 minutes. Evaluating R.H1+, we find several analyses run longer than the desired 60 seconds. In G2.E1, 14/21 repositories run for less than 60 seconds. In G3.E1 (without whitelist), 18/21 repositories run less than 60 seconds. Consequently, we cannot accept the hypothesis R1.H1+. In summary, we can answer RQ2 by concluding that the runtime of the Driftool is acceptable for practical usecases.

Apart from evaluating the hypothesis, we also assessed the influence of the implemented optimizations on the runtime. Figures 15 and 16 show the setup and analysis times for two large repositories. The results display two extreme cases. The *vscode* repository shows a clear decrease in runtime with a higher merge parallelization. The *TypeScript* repository shows no such trend. Both repositories show equal setup times per whitelist configuration. Overall, the runs without

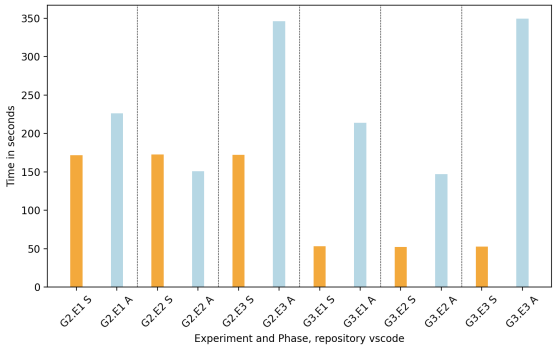

FIGURE 15. Example of a repository's runtime split into setup and analysis intervals. This case shows decreased runtime by increasing merge parallelization.

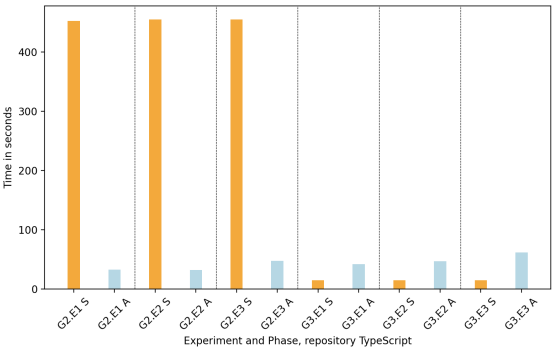

FIGURE 16. Example of a repository's runtime split into setup and analysis intervals. This case shows no benefit of increasing merge parallelization.

a whitelist are faster, which is also supported by Table 3. The effects of merge parallelization on runtime deviate from our initially expected benefits. However, the results are explainable retrospectively. Both repositories comprise a large number of files but differ in the number of active branches. *TypeScript* comprised 12 active branches and *vscode* 79 active branches on Feb 26. 2024. The results suggest that the implemented parallelization only benefits very large branch numbers. Looking at the runtime results in Table 3, we even find repositories with a higher runtime for the runs with more threads. The reason is that the creation of repository copies per thread is time-intensive. If the thread count is high but the number of merges is low, this overhead outweighs the parallelization benefit.

VIII. LIMITATIONS AND THREATS TO VALIDITY

A. STUDY DESIGN

The study design is a potential source of systematic errors and bias. The following paragraphs discuss which threats we identified and how they may have influenced the results.

1) TIME

All results derived from a limited sample over a limited period are naturally limited in their generalizability. The specific period of our study might influence the results.

TABLE 3. Runtime results for experiments groups G2 and G3 The table shows the average analysis runtime per run, i.e., day. The runtime was measured for the analyses of the five days from Feb 26. 2024 until Mar 01. 2024. The runtime is shown in seconds. The standard deviation is shown in parentheses. The runtime measurement started and ended with the start/termination of the Driftool application within the execution environment. The setup of the environment (docker container) is not included but negligible.

Repository	G2.E1	G2.E2	G2.E3	G3.E1	G3.E2	G3.E3
datashaper	15.66 (0.69)	16.67 (0.65)	17.03 (0.75)	5.73 (0.20)	6.63 (0.31)	7.29 (0.08)
PowerToys	74.62 (3.95)	73.09 (3.51)	78.85 (4.30)	24.71 (1.81)	23.85 (0.94)	35.23 (1.94)
semantic-kernel	12.37 (0.65)	12.12 (0.57)	13.38 (0.73)	8.51 (0.23)	8.55 (0.23)	10.37 (0.47)
terminal	73.66 (4.68)	70.03 (4.11)	77.10 (5.16)	18.87 (1.29)	16.63 (0.99)	23.31 (1.51)
TypeScript	484.91 (28.99)	486.46 (28.72)	502.03 (29.91)	56.80 (3.59)	61.96 (3.18)	76.39 (4.27)
vscode	397.85 (27.32)	323.41 (20.25)	518.13 (38.04)	266.68 (20.14)	199.56 (14.56)	401.80 (31.37)
lexical	7.36 (0.10)	7.51 (0.06)	8.00 (0.09)	5.14 (0.05)	5.21 (0.20)	5.59 (0.24)
Rapid	11.19 (0.04)	11.80 (0.41)	12.33 (0.18)	8.09 (0.36)	10.59 (0.28)	11.08 (0.26)
react-native	99.84 (6.50)	85.30 (4.96)	116.28 (6.35)	65.15 (4.92)	53.90 (2.48)	83.10 (5.43)
react	15.97 (0.51)	16.82 (0.63)	19.42 (0.64)	13.64 (0.39)	14.25 (0.52)	16.57 (0.33)
docusaurus	16.31 (1.33)	16.80 (1.24)	17.25 (1.34)	6.01 (0.36)	7.28 (0.23)	8.53 (0.34)
dokka	21.88 (1.42)	21.91 (1.32)	25.23 (1.51)	13.28 (0.91)	13.30 (0.44)	16.71 (0.97)
kotlinx-datetime	1.91 (0.11)	1.93 (0.18)	2.01 (0.13)	1.79 (0.14)	1.80 (0.17)	1.91 (0.11)
kotlinx.coroutines	13.86 (0.36)	12.85 (0.35)	15.36 (0.53)	9.04 (0.40)	8.29 (0.20)	10.70 (0.42)
kotlinx.serialization	5.66 (0.12)	5.75 (0.19)	5.95 (0.15)	2.91 (0.16)	2.94 (0.15)	3.12 (0.11)
dataframe	10.64 (0.10)	10.78 (0.13)	11.40 (0.06)	10.52 (0.13)	12.54 (0.21)	13.91 (0.36)
electron	63.28 (5.07)	59.95 (4.62)	66.80 (5.30)	24.71 (2.79)	19.72 (1.78)	31.76 (3.14)
julia	144.70 (5.14)	111.95 (4.05)	186.04 (11.95)	121.33 (4.26)	87.66 (3.05)	171.28 (11.87)
keras	2.12 (0.12)	2.25 (0.23)	2.44 (0.26)	2.05 (0.13)	2.23 (0.33)	2.34 (0.20)
angular	42.65 (0.48)	45.42 (0.34)	45.75 (0.58)	17.78 (0.37)	24.04 (0.35)	24.56 (0.56)
bootstrap	4.21 (0.76)	4.76 (1.01)	5.07 (0.89)	3.74 (0.54)	4.37 (0.84)	5.10 (0.64)

We acknowledge that analyzing the repositories over any different period would have led to different branch-drift results. However, the concrete values of single drifts are of no concern for our study as we were solely interested in the behavior of relative changes in the branch-drift and their comparison to relative changes in other metrics. Analyzing the repositories over a longer time might have led to a more generalizable result, e.g., we may have missed important events in the repositories. However, a period of three months is adequate for this study. The results support this argument. The drift charts show different hotspots. These hotspots are not too frequent but also not rare. Furthermore, most repositories have multiple releases over the period, indicating that we covered multiple development phases.

2) REPOSITORY SELECTION

We analyzed 21 repositories. As this is a rather small number, one cannot infer general conclusions with confidence. The analyzed repositories might be biased toward a certain outcome. We generally cannot rule out this possibility, but a large bias is unlikely as the results showed high diversity. The repositories vary in purpose, size and activity. The results showed no obvious type- or organization-specific behavior. However, analyzing a set of 21 different repositories will lead to quantitatively different results. Nevertheless, as long as a repository falls within the scope of the branch-drift metric, we are confident that the results provide equally meaningful insights as provided by the analyzed samples. What cannot be inferred from our experiment is the behavior of repositories not fitting the scope of the branch-drift metric as we did not analyze such repositories. Unless this aspect is covered by future research, we advise against using the branch-drift metric on repositories outside the scope of the metric. Furthermore, our experiment design does not allow us to infer any information about how a project's internal structure influences the drift values and trends. This question remains open for future research. However, we were able to demonstrate that the inclusion or exclusion of non-source-code files had no relevant influence on the drift trends.

3) CONFIGURATIONS

The analysis configuration might not be adequate for all repositories. For example, the limit of 60 days of activity for a branch to be considered active may be too short or too long for some repositories. The configured file whitelists might be too restrictive or too permissive. As we are no insiders of the repositories, it is hard to find a more adequate configuration. We cannot rule-out biased results due to our configuration decisions. To get insights into the influence of the whitelist, we conducted different experiment setups. They show no anomalies as discussed in Section VII. Overall, we argue that misconfigured analysis parameters would visually manifest in the presented results. For example, considering too many inactive branches would lead to a high drift baseline. The inclusion of to few branches would lead to a high variance in the results. We argue that the configuration is adequate as we observe stable and plausible results. However, we do not exclude the possibility of better-suited configurations providing more accurate or useful results.

4) INTERPRETATION OF THE RESULTS

The visual interpretation of the drift charts may be biased and subjective. This concerns the acceptance of R1.H1 and R1.H2. The hypotheses R2.H1 and R2.H1+ are accepted based on quantitative results and are not subject to interpretation bias. We acknowledge that it is unavoidable that

the qualitative interpretation is, to some extent, subjective. However, we argue that a possible interpretation bias is irrelevant to the drawn conclusions in this work's evaluation. The purpose of the analysis was not to find specific significant events nor to show specific drift values. The purpose was to show, discuss, and understand branch-drift and its relation to common activity metrics. This analysis is based on reliable quantitative data. Even if the reader disagrees with parts of our interpretation, this work still contributes its results based on the provided data series. All manually analyzed data series and drift charts are published as part of the supplementary material. Thus, other researchers can assess and evaluate the results. In summary, we argue that a possible interpretation bias does not weaken the scientific contributions of this work.

B. TOOLS AND EXECUTION

Errors in the tool arrangement, i.e., the data acquisition, processing, and analysis, can lead to systematically wrong results. We tried to counter this by several measures for quality assurance.

1) DATA ACQUISITION

The data acquisition is comprised of two parts: the cloning of the repositories and the fetching of the commit and release activities. Cloning all repositories via ''git clone'' was triggered manually on each working day. There exist deviations in the cloning time. This is of no concern, as we analyze data over three months. The scripts for fetching the commit and release activity may be error-prone or lead to wrong or unstable results. Additionally, the time of execution may have influenced the results. To counter any potential errors, we took several measures. The scripts to fetch the commit and release activity were executed on the 5th of December, 2024. We choose the execution point to be more than six months after the investigation period's last day. Thus, we are confident that the data is stable and reliable. We tested the scripts on selected repositories, where we manually checked the results. We detected no anomalies. We also executed the fetching process twice within December 2024 and compared the results. We did not detect any differences. We are confident that the fetched data and its acquisition are no error-sources.

2) DRIFTOOL

Section V describes the Driftool and its known limitations. Although we tested the Driftool under various conditions, it always remains possible that the implementation contains errors. All presented drift results rely on the correct working of the Driftool. To verify the Driftool apart from unit testing, we purposefully designed the experiments to include verification runs. We detected no irregularities comparing the experiment results of G1.E1 with G1.E2 and G2.E3 with G2.E4. Checks of G1-runs against the partial results from G2 and G3 also showed no irregularities. Also, the number of threads has shown no influence on the drift results. The only exception is the comparison of the drift values from the symmetrical (merges in both directions and average) and the asymmetrical (only one direction) runs. Deviations within the magnitude of less than 1% were detected for the drift (lines) of 5 repositories. We assume that Git's merge conflict reporting causes these deviations. We consider the deviations negligible. In conclusion, we are confident that the Driftool works as intended and that the results are reliable. We base this confidence on our tests, verification runs, and the stability and well-behavior of the gathered results.

3) POSTPROCESSING

Many data files were generated during the data acquisition and Driftool analysis runs. The CSV and JSON files were postprocessed by Python scripts to collect, combine, compare, and plot the data. All these scripts are a source of potential errors and thus threaten the validity of the results. We approached this threat cautiously, as some of these scripts are quite complex, and it is hard to detect anomalies within plain data series by eye. First, we executed all scripts several times to outrule indeterministic errors. Second, we cross-checked redundant data series to detect possible anomalies. Third, we manually checked multiple data points across the data series and sources. Fourth, we redundantly implemented the most vulnerable part of the postprocessing twice (combining the activity time series and Driftool results into one). Comparing the results of both implementations showed no differences. In conclusion, we are confident that the postprocessing contains no programming errors.

IX. REPRODUCIBILITY/SUPPLEMENTARY MATERIAL

We provide all data, scripts, and tools used in this work to enable reproducibility and reusability of our results. The supplementary material is provided on ZENODO under [52]. Among others, the supplementary material contains:

- Copies of the developed software tools, including the *Driftool* and the scripts for conducting the experiments.
- The drift reports of the 21 analyzed repositories for each day of the experiment interval.
- All generated visualizations from our analysis, including the drift charts of all analyzed repositories.

Continuous releases of the Driftool are openly available on GitHub. The Github repository contains further information on deploying and using the Driftool. Due to its size, the dataset of cloned repositories is not included in the online supplementary material. The repository dataset is archived in a long-term research data storage at the authors' research facility. Access to the repository dataset can be requested from the authors for research and validation purposes.

X. CONCLUSION

A. SUMMARY

Inconsistencies in the form of conflicting changes are a major problem in software development. From an organizational perspective, one cannot know about all individual merge conflicts to assess the potential consistency problems. Instead,

a metric is needed to quantify the existing inconsistencies in an easy-to-analyze numeric value. This work investigated the suitability of *branch drift* as a metric for consistency measurement in software repositories. The branch-drift metric is based on the notion of *Drift* proposed by Kegel et al. in [15]. This work extended the concept of drift, considering the structures and conditions in software development. Experiments based on a dataset of 21 repositories cloned over three month were conducted to show and evaluate the metric's properties. The branch drift results showed well-behaved and visually easy-to-interpret results over time. This raises the potential of the branch-drift metric to become a standard metric for consistency measurement in software repositories. If used continuously, the branch-drift metric could help identify potential merge problems before they occur and update the development team about the current state of the repository. Consequently, timely actions could be taken to keep the repository's consistency in check. Ultimately, this results in a more predictable software development process and a more efficient use of resources. This work contributes the evaluation of the branch-drift metric on a dataset of 21 repositories. Furthermore, this work contributes the implementation of the *Driftool* for computing the metric. Lastly, we contribute all data and code used in the experiments for further research.

B. FUTURE WORK

This work evaluated the branch-drift metric from an ''outside'' perspective. This means we had no deep technical or organizational insights into the analyzed repositories. We also did not confront developers of the analyzed repositories with the results. Future work must investigate the metric from an ''inside'' perspective. Therefore, studies must be conducted to evaluate the acceptance of the metric and practical usefulness within a development team. Furthermore, our experiment setup does not allow for inferring causal connections between the project structure and the branch-drift metric. It is not clear which design or development decisions lead to a high or low branch-drift value. Therefore, future work must investigate this possible connection.

REFERENCES

[1] R. Conradi and B. Westfechtel, ''Version models for software configuration management,'' *ACM Comput. Surv.*, vol. 30, no. 2, pp. 232–282, Jun. 1998.
[2] C. Walrad and D. Strom, ''The importance of branching models in SCM,'' *Computer*, vol. 35, no. 9, pp. 31–38, Sep. 2002.
[3] W. Zou, W. Zhang, X. Xia, R. Holmes, and Z. Chen, ''Branch use in practice: A large-scale empirical study of 2,923 projects on GitHub,'' in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Jul. 2019, pp. 306–317.
[4] J. C. C. Ríos, S. M. Embury, and S. Eraslan, ''A unifying framework for the systematic analysis of git workflows,'' *Inf. Softw. Technol.*, vol. 145, May 2022, Art. no. 106811.
[5] D. Perry, H. Siy, and L. Votta, ''Parallel changes in large-scale software development: An observational case study,'' *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 3, pp. 308–337, Jul. 2001.
[6] G. Ghiotto, L. Murta, M. Barros, and A. van der Hoek, ''On the nature of merge conflicts: A study of 2,731 open source Java projects hosted by GitHub,'' *IEEE Trans. Softw. Eng.*, vol. 46, no. 8, pp. 892–915, Aug. 2020.
[7] H. L. Nguyen and C.-L. Ignat, ''An analysis of merge conflicts and resolutions in git-based open source projects,'' *Comput. Supported Cooperat. Work (CSCW)*, vol. 27, nos. 3–6, pp. 741–765, Dec. 2018.
[8] S. Apel, O. Leßenich, and C. Lengauer, ''Structured merge with auto-tuning: Balancing precision and performance,'' in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.* New York, NY, USA: ACM, Sep. 2012, pp. 120–129.
[9] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, ''The impact of structure on software merging: Semistructured versus structured merge,'' in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1002–1013.
[10] C. Brindescu, I. Ahmed, R. Leano, and A. Sarma, ''Planning for untangling: Predicting the difficulty of merge conflicts,'' in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 801–811.
[11] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, ''Predicting merge conflicts in collaborative software development,'' 2019, *arXiv:1907.06274*.
[12] M. L. Guimarães and A. R. Silva, ''Improving early detection of software merge conflicts,'' in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 342–352.
[13] C. Seaman and Y. Guo, ''Measuring and monitoring technical debt,'' in *Advances in Computers*, vol. 11. Amsterdam, The Netherlands: Elsevier, 2011, pp. 25–46.
[14] P. Kruchten, R. L. Nord, and I. Ozkaya, ''Technical debt: From metaphor to theory and practice,'' *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov. 2012.
[15] K. Kegel, S. Götz, R. Marx, and U. Aßmann, ''A variance-based drift metric for inconsistency estimation in model variant sets,'' in *Proc. J. Object Technol.*, Jul. 2024, vol. 23, no. 3, pp. 1–14.
[16] G. Widmer and M. Kubat, ''Learning in the presence of concept drift and hidden contexts,'' *Mach. Learn.*, vol. 23, no. 1, pp. 69–101, Apr. 1996.
[17] F. Bayram, B. S. Ahmed, and A. Kassler, ''From concept drift to model degradation: An overview on performance-aware drift detectors,'' *Knowl.-Based Syst.*, vol. 245, Jun. 2022, Art. no. 108632.
[18] K. Feichtinger, K. Kegel, R. Pascual, U. Aßmann, B. Beckert, and R. Reussner, ''Towards formalizing and relating different notions of consistency in cyber-physical systems engineering,'' in *Proc. ACM/IEEE 27th Int. Conf. Model Driven Eng. Lang. Syst.* New York, NY, USA: ACM, Sep. 2024, pp. 915–919.
[19] T. Wuensche, A. Andrzejak, and S. Schwedes, ''Detecting higher-order merge conflicts in large software projects,'' in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 353–363.
[20] C. Tinnes, W. Rössler, U. Hohenstein, T. Kühn, A. Biesdorf, and S. Apel, ''Sometimes you have to treat the symptoms: Tackling model drift in an industrial clone-and-own software product line,'' in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA: ACM, Nov. 2022, pp. 1355–1366.
[21] M. Staron, ''Metrics for software design and architectures,'' in *Automotive Software Architectures*. Cham, Switzerland: Springer, 2021, pp. 215–233.
[22] H. Nakai, N. Tsuda, K. Honda, H. Washizaki, and Y. Fukazawa, ''Initial framework for software quality evaluation based on ISO/IEC 25022 and ISO/IEC 25023,'' in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Aug. 2016, pp. 410–411.
[23] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. Boca Raton, FL, USA: CRC Press, 2014.
[24] S. R. Chidamber and C. F. Kemerer, ''A metrics suite for object oriented design,'' *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
[25] P. Oman and J. Hagemeister, ''Construction and testing of polynomials predicting software maintainability,'' *J. Syst. Softw.*, vol. 24, no. 3, pp. 251–266, Mar. 1994.
[26] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister, ''Constructing and testing software maintainability assessment models,'' in *Proc. 1st Int. Softw. Metrics Symp.*, pp. 61–70, Jan. 2002.
[27] J. F. Ramil and M. M. Lehman, ''Metrics of software evolution as effort predictors—A case study,'' in *Proc. Int. Conf. Softw. Maintenance (ICSM)*, Oct. 2000, pp. 163–172.
[28] J. R. Larus and T. Ball, ''Rewriting executable files to measure program behavior,'' *Softw., Pract. Exper.*, vol. 24, no. 2, pp. 197–218, Feb. 1994.
[29] T. Ball, ''The concept of dynamic analysis,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, Nov. 1999.
[30] H. Zhu, P. A. V. Hall, and J. H. R. May, ''Software unit test coverage and adequacy,'' *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[31] K. Yamamoto, M. Kondo, K. Nishiura, and O. Mizuno, "Which metrics should researchers use to collect repositories: An empirical study," in *Proc. IEEE 20th Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, Dec. 2020, pp. 458–466.

[32] J. Byun, S. Rhew, M. Hwang, V. Sugumara, S. Park, and S. Park, "Metrics for measuring the consistencies of requirements with objectives and constraints," *Requirements Eng.*, vol. 19, no. 1, pp. 89–104, Mar. 2014.

[33] K. Mu, Z. Jin, R. Lu, and W. Liu, "Measuring inconsistency in requirements specifications," in *Symbolic Quantitative Approaches to Reasoning with Uncertainty*, vol. 3571, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. P. Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and L. Godo, Eds., Berlin, Germany: Springer, 2005, pp. 440–451.

[34] T. Berger and J. Guo, "Towards system analysis with variability model metrics," in *Proc. 8th Int. Workshop Variability Model. Softw.-Intensive Syst.* New York, NY, USA: ACM, Jan. 2014, pp. 1–8.

[35] S. El-Sharkawy, N. Yamagishi-Eichler, and K. Schmid, "Metrics for analyzing variability and its implementation in software product lines: A systematic literature review," *Inf. Softw. Technol.*, vol. 106, pp. 1–30, Feb. 2019.

[36] C. I. M. Bezerra, R. M. C. Andrade, and J. M. S. Monteiro, "Measures for quality evaluation of feature models," in *Software Reuse for Dynamic Systems in the Cloud and Beyond*. Cham, Switzerland: Springer, 2014, pp. 282–297.

[37] S. J. Niepostyn and W. B. Daszczuk, "Entropy as a measure of consistency in software architecture," *Entropy*, vol. 25, no. 2, p. 328, Feb. 2023.

[38] F. J. Furrer, *Future-Proof Software-Systems*. Cham, Switzerland: Springer, 2019.

[39] C. Lilienthal, *Sustainable Software Architecture*. Heidelberg, Germany: Dpunkt Verlag, 2019.

[40] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: Survey and empirical study," *Automated Softw. Eng.*, vol. 25, no. 2, pp. 279–313, Jun. 2018.

[41] B. Shen and N. Meng, "ConflictBench: A benchmark to evaluate software merge tools," *J. Syst. Softw.*, vol. 214, Aug. 2024, Art. no. 112084.

[42] Object Management Group. (2024). *Unified Modeling Language UML Reference Documentation 2.5.1*. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML

[43] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 2, no. 1, pp. 31–57, 1992.

[44] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, "A feature-based survey of model view approaches," *Softw. Syst. Model.*, vol. 18, no. 3, pp. 1931–1952, Jun. 2019.

[45] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, Eur. J. for Informat. Prof.*, vol. 9, no. 2, pp. 29–34, Apr. 2008.

[46] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, "Semistructured merge: Rethinking merge in revision control systems," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.* New York, NY, USA: ACM, Sep. 2011, pp. 190–200.

[47] I. Borg and P. J. Groenen, *Modern Multidimensional Scaling: Theory and Applications*. Cham, Switzerland: Springer, 2005.

[48] H. Toutenburg and C. Heumann, *Deskriptive Statistik: Eine Einführung in Methoden Und Anwendungen Mit R Und SPSS*. Cham, Switzerland: Springer, 2008.

[49] K. Schwaber, "Scrum development process," in *Business Object Design and Implementation*, J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, Eds., London, U.K.: Springer, 1997, pp. 117–134.

[50] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. London, U.K.: Pearson Education, 2007.

[51] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[52] K. Kegel, "Software tools and dataset/supplementary material for 'branch drift: A visually explainable metric for consistency monitoring in collaborative software development,'" Zenodo, Mar. 2025, doi: 10.5281/zenodo.15005698.

KARL KEGEL received the bachelor's and master's degrees in computer science from Dresden University of Technology, Germany, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree. He was a Research Associate with the Chair of Software Technology, Dresden University of Technology, in 2021. The key topics of his ongoing Ph.D. are software merging, collaborative software development, and software metrics, particularly from a model-driven standpoint. His further research interests include human factors in software engineering, software evolution, and reactive self-aware systems.

SEBASTIAN GÖTZ received the Ph.D. degree from Dresden University of Technology, in 2013, working on multi-quality auto-tuning. He is currently a Tenured Faculty Member with Dresden University of Technology, Germany. His research interests include model-driven software engineering, self-optimizing software systems, energy-efficient software systems, and software engineering for robotics. He is part of several scientific communities, in particular in the program committees of the International Conference on Model-Driven Engineering Languages and Systems (MODELS), the International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), and the International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). He organized several international workshops, including the annual international workshop on models@run.time, since 2012, and the international workshop on model-driven robot software engineering (MORSE).

UWE AßMANN received the Ph.D. degree in compiler optimization and the Habilitation degree in invasive software composition, a software reuse technology for arbitrary programs and modeling languages. He is an internationally recognized expert in model-driven software engineering (MDSD), in particular for embedded systems and collaborative robotics. He has been a Professor of software engineering with the Faculty of Computer Science, TU Dresden, since 2004, and was the Dean of the Faculty, from 2016 to 2021. He is well known for his work in metamodeling, adaptive petri nets, and context-role-oriented modeling. He has successfully supervised more than 25 Ph.D. students on topics in MDSD. More than 100 publications on a broad spectrum of software engineering topics have resulted. He has been a Mentor of startups (so far eight), also of https://www.wandelbots.com. His research interests include software composition, software- and modeling languages, attribute grammars, grammar-based software engineering, robotic software engineering, and software quality. He is a member of the DFG EXC Center of Tactile Internet, DFG SFB/TRR 339 "Digital Twin of the Road of the Future," DFG SFB 1618 "Consistent View-based Development of Cyber-Physical Systems," and the DFG Research Training Group "AirMetro." From 2016 to 2024, he has been a main reviewer for software Engineering in the Fachkollegium Informatik of DFG.