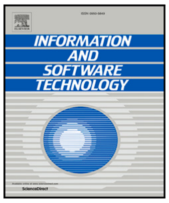
Information and Software Technology 145 (2022) 106811

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



A unifying framework for the systematic analysis of Git workflows

Julio César Cortés Ríos [a,*], Suzanne M. Embury [a], Sukru Eraslan [b]

[a] Department of Computer Science, University of Manchester, Manchester M13 9PL, UK
[b] Middle East Technical University, Northern Cyprus Campus, 99738 Kalkanlı, Güzelyurt, Mersin 10, Turkey

ARTICLE INFO

Keywords:
Git
Workflows
Feature-based modelling
Version control
Branching

ABSTRACT

Context: Git is a popular distributed version control system that provides flexibility and robustness for software development projects. Several workflows have been proposed to codify the way project contributors work collaboratively with Git. Some workflows are highly prescriptive while others allow more leeway but do not provide the same level of code quality assurance, thus, preventing their comparison to determine the most suitable for a specific set of requirements, or to ascertain if a workflow is being properly followed.
Objective: In this paper, we propose a novel feature-based framework for describing Git workflows, based on a study of 26 existing instances. The framework enables workflows' comparison, to discern how, and to what extent, they exploit Git capabilities for collaborative software development.
Methods: The framework uses feature-based modelling to map Git capabilities, regularly expressed as contribution guidelines, and a set of features that can be impartially applied to all the workflows considered. Through this framework, each workflow was characterised based on their publicly available descriptions. The characterisations were then vectorised and processed using hierarchical clustering to determine workflows' similarities and to identify which features are most popular, and more relevant for discriminatory purposes.
Results: Comparative analysis evidenced that some workflows claiming to be closely related, when described and then characterised, turned out to have more differences than similarities. The analysis also showed that most workflows focus on the branching and code integration strategies, whilst others emphasise subtle differences from other popular workflows or describe a specific development route and are, thus, widely reused.
Conclusion: The characterisation and clustering analysis demonstrated that our framework can be used to compare and analyse Git workflows.

1. Introduction

1.1. Background

A *Git workflow* is a collection of guidelines governing how software developers will collaborate through Git [1]. Workflows specify how work should be organised using branches and local clones, how and when work can be integrated, what quality gates are applied to the code and when, and how finished versions of the system are packaged for release. Thus, consistency and compliance with institutional standards and policies are at risk if there is no workflow defined for a development project. By following workflow rules, software developers may collaborate more effectively as they may prevent the problems that could be caused by inconsistent working styles. Besides this, if a particular workflow is used, it would be simpler to add a new member to the development team as s/he will require to apply specific rules.

The importance, and complexity, of choosing an adequate workflow stems from the development process needed for a particular project. For instance, some workflows provide strong protection for the live code base, with complex branching strategies that keep code changes isolated until they have passed through multiple layers of quality checks. Others emphasise fast delivery cycles, allowing fixes and new features to reach the live code quickly, or else attempt to minimise time spent in "integration hell" when long-running branches with many code changes must be merged [2]. Each development team must choose a workflow that suits both their preferred working practices and their code quality goals. This usually involves some trade-offs: choosing a less protective workflow if it is inexpensive and easy to apply, or adopt a more cumbersome branching requirements if it helps ensure that only high quality code is integrated into releases [3]. A wrong choice when deciding which workflow to follow for a particular project may impact negatively on how the developers use their time (as it could be wasted in repetitive instructions or clarifications), or affect the production of

* Corresponding author.
E-mail addresses: juliocesar.cortesrios@manchester.ac.uk (J.C. Cortés Ríos), suzanne.m.embury@manchester.ac.uk (S.M. Embury), seraslan@metu.edu.tr (S. Eraslan).

https://doi.org/10.1016/j.infsof.2021.106811
Received 23 March 2021; Received in revised form 13 December 2021; Accepted 19 December 2021
Available online 10 January 2022

clean code by imposing vague guidelines around quality checks before merging a particular feature.

1.2. Research problem

The **research problem** that this work aims to address considers that there are nowadays a number of actively-used workflows, and several that are fitted for specific purposes, each espousing a particular approach to collaborative coding. Since these workflows have emerged from practice, they are typically described informally, in a way that makes their advantages or disadvantages hard to discern, and not straightforward to follow in practice. These workflows are frequently specified in varying levels of detail using different terms, scopes and visual representations. While the blogosphere is full of critical comparisons between different pairs of workflows [4–6], the lack of a common framework defining workflows for software development precludes a systematic and efficient evaluation of the available options.

The kind of differences between the workflow descriptions range from the use of singular terminology (e.g., feature, topic or change to designate a dedicated line of development for a specific piece of the software product) to varied level of detail and formalisation when specifying the guidelines (e.g., a partial and informal list of instructions vs. a detailed set of directives using standard notation).[1] Other guidelines may include visual aids, such as, branching diagrams, figures representing user restrictions or images of the steps the contributor is expected to follow. Thus, it is hard to find a simple mechanism to compare these specifications to, for instance, evaluate the benefits of a given approach against an alternative.

1.3. Research questions

For the aforementioned problem, our research aims to conceptualise and create a unified framework that can help in satisfying the following **research questions (RQ)**:

RQ1. What is the range and characteristics of the Git workflows that have been proposed to date?
RQ2. Can we define a set of features that can be used to consistently describe all Git workflows identified for **RQ1**?
RQ3. Given the features denoted in **RQ2**, can we use them to characterise the Git workflows to enable the identification of similarities and differences between them?
RQ4. Given the characterisations from **RQ3**, can we compare and aggregate the Git workflows to identify families with similar features?

1.4. Proposed solution

As a research output, in this paper we propose a common framework, based on feature-based modelling [7], for specifying Git workflows to address the aforementioned research problem and associated questions. To the best of our knowledge, this is the first such framework to be proposed, either in the academic or non-academic literature. Our proposal is based on a detailed study of 26 Git workflows whose descriptions are publicly available.

The framework consists of a hierarchical collection of features, covering such aspects of the collaborative development process as repository setup, file management, syncing, branching and code integration strategies, and additional features.

As well as allowing the comparison and analysis of workflows, our framework has been applied to all the workflows considered in this study, to show not only the similarities and differences between these workflows, but also to support the detection of subtle properties, such as: which workflows are clearly defining different development approaches and which can be used to build other, more complex, workflows. It is important to notice that, although authors of new workflows can use our framework to specify their guidelines, it can also be used by them and any Git user as a condensed reference of those practices and features that are commonly used in Git workflows, which in turn has the potential of improving the way such development guidelines are specified and even followed.

1.5. Contributions and paper structure

This paper presents the following **contributions**:

- the first comprehensive survey of existing workflows with an analysis of their components, taken from their specifications, to support the characterisation of individual Git workflows (Section 4);
- a feature-based framework for the characterisation, comparison and clustering of Git workflows (Section 5);
- the first systematic characterisation of Git workflows using a common framework (Section 6); and
- the application of the framework for the analysis of the workflows considered for this study (Section 7).

The remainder of this paper is structured as follows, Section 2 explores the associated literature, Section 3 describes our research methodology, and Section 8 presents the limitations of our study, its impact and future work.

2. Literature review

Despite the popularity of Git as a version control system, and the importance of choosing the right workflow for project success (e.g., [8,9]), even so, these workflows have received scant attention in the scientific and grey [10] literature.

The academic literature on Git workflows mainly focuses on proposals for new workflows, or recommendations for adaptations of existing workflows to fit new settings (Section 2.1).

Most of the workflow descriptions used in this work are from the grey literature (i.e., materials generated outside of the scientific community; Section 2.2). Several comparisons between workflows, with varying levels of rigour, are also available in those materials. To the best of our knowledge, there are no other studies attempting to describe workflows using a common framework, for their systematic characterisation and analysis.

2.1. Scientific literature

Most popular workflows, such as GitFlow [11] or GitHub Flow [12], are described either informally (on blogs, repository documentation or personal web pages) or using proprietary terminology and presentation layouts, but it is unusual for workflows to be described in the scientific literature, except for the purposes of providing background to a description of some related topic (e.g., [13]). GitWaterFlow is a rare counter-example: Rayana et al. proposed a workflow based on automating parts of the development process, with the aim of reducing development disruption, coping with non-trivial feature development, and supporting large development teams while maintaining the quality of the software [14].

Regardless of how these workflows are presented, in a few cases, authors have proposed adaptations of existing workflows to fit specific development contexts. Hellebrand et al. propose a customisation of the popular GitFlow workflow for use in product line engineering (PLE) [15]. The resulting workflow specifies two different GitFlow-based branching models: one for organising work on the stable product line assets, and another for handling the tasks associated with the

[1] https://tools.ietf.org/html/rfc2119.

product variants created for specific user communities. Similarly, Montalvillo and Díaz adapted GitFlow by proposing several new branch types specialised for developing product line assets and product variants, and for propagating fixes and features between them [16]. In another domain, Gary et al. have described a branching strategy based on feature branches that has been used successfully on a safety-critical open source health project [17].

2.2. Grey literature

An increasing number of resources from the grey literature aim to define and compare Git workflows, with varied level of success in their attempts to present the features of these workflows, or their similarities and differences in a clear and concise manner.

For instance, in terms of defining workflows, the description of Git Common Flow [18] provides a clear specification of its guidelines, embedded in a structure that makes use of well-known standards. Following a more informal approach, the description of Simple Git Flow [19] also provides clear instructions but presenting them as steps that the user should follow, with some examples and using again informal language. In another example, the definition found for Skullcandy's workflow [20] presents the guidelines as a list of points, that make use of a simple branching diagram but describes just a minimal portion of the development aspects, leaving to the user the decision of how to interpret those rules in practical terms.

Regarding the comparison of Git workflows, most of the comparisons are presented following heterogeneous structures, terminology and visual aids. Some of the related resources present the instructions with an appropriate level of rigour, such as, those presented in [21–24], but even those, just provide limited information that do not support their systematic comparison, given that just a few features are analysed in detail. Most of the resources on workflow comparison are presented in an unstructured and informal way, sometimes using helpful examples and diagrams, but at the same time leaving considerable room for interpretation of the advantages and disadvantages between the compared workflows. As an example of these resources we have [20,25–29]. And finally, there are also some comparisons that, based on their succinctness or lack of sufficient explanation, cannot be practically used to take any decisions regarding which workflow to use, such as [30,31].

Most of these works, focus on comparing a small number of workflows (between two and five), and even from those that provide an informal analysis, this can be hardly translated into a systematical comparison, to help determine how or when these workflows could be used, information that could be critical as explained in Section 1.2.

The works outlined in this section cover different topics associated with the definition or comparison of workflows, which are the centre of our study, however, none of them, to the extent of our knowledge, focuses on the generic description of workflows to support their systematic characterisation and analysis.

3. Methodology

In this Section, we provide a description of the data and methods used to address our research questions.

3.1. Survey on Git workflows

To answer **RQ1** and discover what types of actively-used workflows there are and which characteristics they have, we conducted a survey on those workflows that are (or have recently been) in active use. Considering that references to Git workflows may exist in multiple locations, such as, blogsphere, personal websites, development repositories, corporate or product documentation, among others; a decision was made to focus on those that can be publicly accessed (i.e., public websites and open software development platforms), to be able to use and analyse such descriptions without breaching confidentiality agreements.

The first step was searching in the following public software development platforms:

- GitHub
- GitLab
- Bitbucket
- SourceForge
- FramaGit
- GitGud
- Pagure

Using the search engines of these platforms, we examined the contents of the markdown files (.md extension), for each repository, filtering out those containing the terms: "workflow" or "branching strategy" or "merging strategy". The search across all platforms returned a list on which 68 different candidate workflows were identified, once the entries were de-duplicated. Multiple fork-levels were not taken into account as we are only looking for unique workflow specifications regardless of the origin of the repository from which the specification is obtained. The next step was to filter out those entries that, although they mention the terms searched, the content was not related to the definition of guidelines for software development (e.g., like those matches where the term "workflow" was used in a sense that does not relate to Git). Workflows for which no documentation could be found (e.g., just the name of the workflow was mentioned) were also removed. After these steps the list of different workflows found in these platforms contained 6 entries.

To complete our survey, we reviewed the scientific literature and other online sources (such as blogs [19,20,25,32] and personal webpages [11,24,33,34]) using a web search engine with the same terms used for the software development platforms. This second exploration identified 20 additional workflows, including three research papers that define a Git workflow for specific purposes [14,15,17].

In total, 26 Git workflows were identified. Their main characteristics and the associated references are listed in Table 1.

3.2. Framework creation

From the survey on Git workflows, we realised that the descriptions we found varied dramatically in their structure, level of detail and completeness, therefore, we needed to find a way to fairly and objectively compare the workflows described.

To answer **RQ2** – and find out if we could generate a list of features to consistently describe Git workflows – we followed three steps, the conceptualisation of a framework based on keywords from the workflow descriptions, the modelling of those keywords using feature-based modelling, and the extraction of the features.

3.2.1. Keyword identification

In the first step, we bring together all the workflow descriptions found in a single document (see the Section Section 8.2 at the end of this document), to facilitate the searches and its analysis. In this step, we read the descriptions looking for words or phrases that give a notion of an instruction for a software development process, such as, "branch off from ...", "integrate ... into ...", "tag", "release", "code review", among others, and also looking at terms commonly used to describe a level of enforcement associated with a requirement, such as, "must", "should", "may" (as explained in RFC2119).

Following the aforementioned process, we obtained from the workflow descriptions a list of keywords that are frequently used to describe guidelines for collaborative software development. We mapped these keywords with the functionality provided by the Git version control system, or supported through the use of external and commonly used tools (e.g., the "tag" keyword with the tag command in Git, the "integrate" keyword with the merge command in Git, or the "issue tracker" keyword with the issue tracking functionality of external tools like GitLab or GitHub), to create a working framework that allows the comparison of Git workflows using common terminology instead of heterogeneous descriptions.

Table 1
Overview of Git Workflows found in the survey.

Workflow	Id	Main characteristics	Refs.
Backcountry	BaC	Development in main. Release and change branch off from main and must be merged back	[20]
BitBucket	BiB	Production and staging branches. Prefers pull requests. Hotfixes directly in production.	[35]
BOINC Flow	BoF	Main is reliable and everyone should branch from it. Development on feature or bugfix branch. Pull requests to merge. Delete feature and bugfix branches once merged.	[36]
Cactus model	Cac	Development in main. Rebase commits instead of changes merged back into main. Cherry-pick from fix branches.	[37]
Centralized	Cen	Commits applied into main branch.	[20,23,25,29,31]
Dictator and Lieutenants	D&L	Development on topic branches and rebase on top of main (from reference). Lieutenants merge developers' topics into their main. Dictator merges lieutenants' main into its own (pushed to reference).	[38]
Feature branch	FBr	Development of new features in change branches.	[23,25,29]
Feature branch w/code reviews	FBC	Code review encouraged through pull requests.	[39]
Fork	Frk	New changes on server copy of reference repository.	[40]
Fork & merge	F&M	Developers have their own forked repository on hosting site. New features merged into original project.	[23,25,29,31]
Git Common	Com	Commits on change branches, rebasing commits into main.	[18]
GitFlow	GiF	Develop branch for integration, and release and fix branches for versioning and bug correction respectively.	[11,20,23,29,31,41]
GitHub	GHF	Main always deployable. Intensive use of pull requests.	[12,20,28]
GitLab	GLF	Feature-driven with change branches and issue tracking.	[28,42,43]
GitWaterFlow	GWF	Inspired on semantic versioning. Tools to automate development. Promotes forward porting.	[14]
IGSTK Flow	IGS	For healthcare industry. Uses change and release branches.	[17]
Maintenance	Mnt	Dedicated hotfix branch for bug correction in production.	[44]
OneFlow	One	Single long-lived main, short-lived: change, release and fix.	[28,32]
PLEFlow	PLE	Two branching models for product and variant levels.	[15]
Release	Rel	Features into main. Fixes cherry-picked to release. No tags.	[29,45,46]
Simple Git	Sim	Main is product line. Rebase commits from main into change to update it and solve conflicts promptly.	[19]
Skullcandy's	Skl	QA performed on dedicated branch. Relies on issue tracking.	[20]
Stash team flow	STF	Develop, release and hotfix branches, no main.	[47]
Three-Flow	Thr	Main, candidate and release branches. Rebase commits from change into main. Uses feature toggles.	[34]
Trunk-based development	TBD	Work on single open branch. Enforceable code style with fast iterations and no usage of pull requests.	[4,48]
WunderFlow	WuF	Features branched off from main, develop for testing and reviewing, and production for installed code. Development in features, epic features (larger), hotfix and release branches.	[49]

3.2.2. *Feature-based modelling*

The second step for answering **RQ2**, was the creation of the framework. For this step we use feature-based modelling [50], which is part of a broader variability modelling approach [51,52] commonly used to describe software product line engineering [7,53,54], but which has also been applied when a subject can be described in terms of its features, and how these features vary on different instances (or versions) of the subject of interest. This approach can be applied in our context because it is a well-known mechanism to describe: aspects of a collection (or family) of related entities, the elements they have in common, and how these entities vary on different instances across this collection.

In our study, the modelling of the features is based on the extracted keywords from the workflow descriptions, representing guidelines to be followed, and how these guidelines map against the functionality provided or supported by Git.

Another benefit of using feature-based modelling to represent the guidelines of Git workflows, is that this type of modelling is frequently translated into an schematic representation, or *feature diagram* [55], that can facilitate the visualisation of the relations and constraints between the features. This type of visualisation was introduced by Kang [56], as part of a methodology called *Feature Oriented Domain Analysis (FODA)*. A feature diagram, is a graph in which the *nodes* are features, and it commonly follows a tree-like structure, representing the hierarchical relationships between the features of a product or other entity.

3.2.3. *Feature extraction*

A user of the framework will then associate the guidelines found in the description of a Git workflow with a feature in our model and associate additional properties, as required by the feature, to translate the informal content from the description into a crisp feature in our framework.

For the third step, each individual statement (content associated with a guideline for software development) was identified and then classified as one the following types:

- Textual
  - Simple statement. Single sentence describing a guideline or property belonging to the current workflow. For example, the statement "*All development happens on the master branch*" (from the Cactus model workflow [37]) would be classified in this way.
  - Composed statement. Two or more sentences that may or may not be written together but convey a single guideline or property for the current workflow. For example, the text "*... the default development branch is called master... This workflow does not require any other branches besides master ...*" (from the Centralized workflow [23]) would be classified as this type of statement.
  - Command example. Instruction that explicitly states (fully or partially) the Git command to be executed to achieve a particular effect. For example, the text "*git pull –rebase*

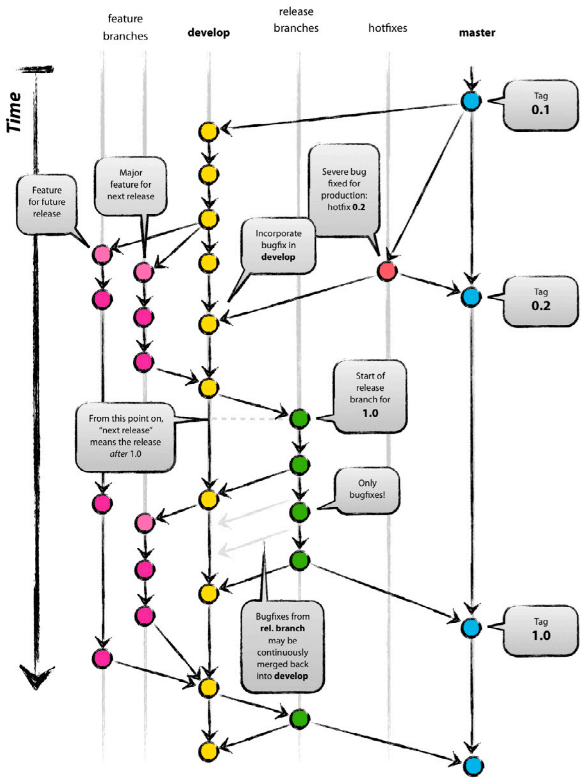

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Fig. 1. Visual content example of a branching diagram describing a Git workflow [11].

origin master" (from the Centralized workflow [23]) would be classified in this way.
  - Inherited feature. Indication that some guideline or property of the current workflow that is explicitly copied or derived from another workflow. For example, the text "*The Forking Workflow typically follows a branching model based on the Git-Flow Workflow*" (from the Fork and Merge workflow [23]) would be classified as this type of statement.

- Visual
  - Branching diagram. Schematic representation normally containing commits, branches and textual aid elements. Example in Fig. 1.
  - Other type of image. Other type of visual representation indicating a development guideline.

3.3. Characterising Git workflows

For the characterisation of Git workflows (**RQ3**), we distributed the work, of reading and identifying which features appeared on which workflow descriptions, among two of the authors of this work, who completed the characterisations independently. A full set of instructions and auxiliary materials were provided (see the Section Section 8.2 at the end of this document), along with the framework itself. A high level of agreement was achieved between the two authors (93.72%) in identifying the features of workflows with the enforcement levels (i.e. recommendation, requirement, etc.) and an un-weighted Cohen's Kappa shows highly moderate agreement (kappa=0.60).[2] The discrepancies were resolved through discussion.

[2] <0 Less than chance agreement, 0.01–0.20 Slight agreement, 0.21–0.40 Fair agreement, 0.41–0.60 Moderate agreement, 0.61–0.80 Substantial agreement, 0.81–0.99 Almost perfect agreement [57].

3.4. Comparing and clustering Git workflows

To answer **RQ4**, the workflow characterisations were then compared in a vectorised form using the following equation to compute the euclidean distance:

$$d(V_{wf_1}, V_{wf_2}) = \sqrt{(V_{wf_1} - V_{wf_1})^2} \qquad (1)$$

where V_{wf_x} is the vectorised form of the characterisation of workflow x.

This distance was used to obtain the degree of similarity between the characterised workflows. Other distance metrics were considered (such as Cosine, Manhattan, Minkowski and Hamming), but the euclidean was selected as it accepts vector elements with value zero, and scales to multiple dimensions.

Finally, we applied *hierarchical clustering*[58] over the vectorised characterisations of the Git workflows to find out how they grouped together when taking into account the features indicated by their characterisations. This process made use of an unsupervised machine learning-based algorithm that uses the euclidean distance between the vectorised characterisations to iteratively build the clusters that aggregate Git workflows based on their similarity.

4. Git workflows

Following the steps described in Section 3.1, we conducted a survey that found the descriptions of 26 Git workflows to respond to our **RQ1**. To the extent of our knowledge, the resulting list from our survey is the first one to include such a number of heterogeneous Git workflows. The result also confirmed that the Git workflows are described using varied terminology, with various levels of rigour, and providing unequal coverage about the aspects that typically conform to a development process.

4.1. List of workflows found

In Table 1, we list the Git workflows found during our survey of scientific and non-scientific literature, indicating the main features, from an overall perspective, and, for each workflow, a list of references to their documentation. This table includes the full name of the workflow, and also a mnemonic (column "Id") for reference purposes.

4.2. Common aspects and concepts in a Git workflow description

From the 26 Git workflows found, we identified the following 6 broad categories under which the guidelines describing these workflows can be aggregated, as illustrated in Fig. 2:

1. **Repository setup and initialisation**. Steps and configurations, required or recommended by a workflow, to prepare the repository for the development process.
2. **File management**. Git commands associated with the staging, labelling or index management of changes, excluding those related with branching and code integration, that are required or recommended by a workflow, and how they are used, as part of the development process.
3. **Syncing**. Synchronisation mechanisms that should be followed by the participants of a software development project.
4. **Branching strategy**. Conventions followed on how branches should be used to partition and organise the work.
5. **Code integration strategy**. Approach, mechanism or procedure followed during the integration of one line of development into another.
6. **Additional features**. Supplementary guidelines or aspects, not covered by other categories, that need to be considered by the participants in a software development project.

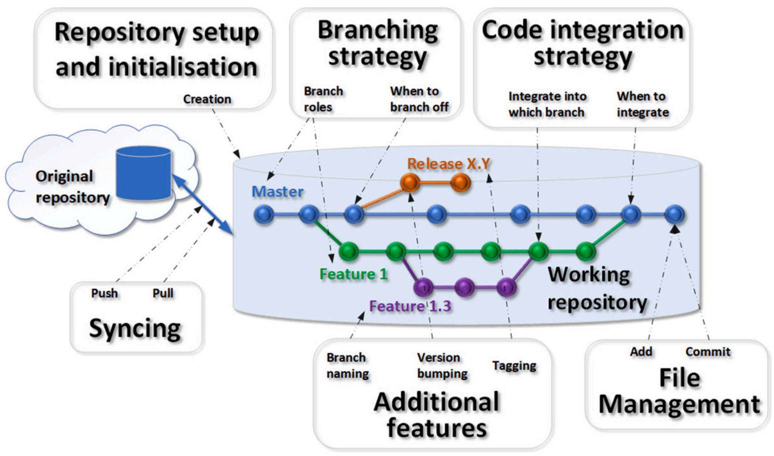


Fig. 2. Categories of guidelines in Git workflows.

Among the concepts, commonly found in the workflow descriptions, were the different roles a line of development can take depending on how it is used during the development process, as indicated in Section 4.2. It is important to note, that a line of development should take only one role and this relation must be maintained throughout the development project.

As can be seen in Fig. 2, work proceeds by dividing the development into separate lines of development, on which different sub-teams work, sometimes in parallel, until the new code is ready to be integrated [59]. The terminology used to describe these development lines varies considerably across the workflow descriptions. Following are the main roles found, in the context of the development process, that these lines of development can take, and that can be used in our framework instead of the variety of development lines frequently used in the workflow descriptions:

i. **Main**. Contains all the changes associated with the last stable release. Associated branches: Master, Trunk.
ii. **Integration**. Allows the integration of multiple changes that are not yet ready to be released. Associated branch: Develop.
iii. **Change**. All work associated with the development of a new feature. Associated branches: Feature, Topic.
iv. **Release**. Isolates all changes for a particular version in preparation for its launch. Associated branches: Release, Production.
v. **Validation**. Isolates a set of changes for revision, testing or demonstration purposes. Associated branches: QA, Test, Preproduction.
vi. **Fix**. Development work needed to correct bugs and other emergency fixes. Associated branches: Fix, HotFix, BugFix.

4.3. Discussion

Responding to **RQ1**, our survey of Git workflows returned 26 workflows that are actively used in real-world development projects, with varied conditions such as the size of the team, the scope among the development process, or the restrictions imposed to the contributors. The diversity in how Git workflows are frequently specified complicates their systematic analysis, such as in the identification of the similarities or differences between pairs of workflows.

After the 26 Git workflows were compiled and studied, it became clear that many of the guidelines rely on subjective notions, such as the concept of "*long-lived branches*" (how long is long?) or the instruction "*to commit as frequently as possible*", that make it hard to come up with a simple way to specify all workflows consistently. These imprecise aspects of the workflow description often contain important information about the workflow; they cannot simply be ignored. Therefore, a hybrid approach is needed that supports both precise and vague instructions to be represented. This is the approach we followed in our framework (our response to **RQ2**), and we acknowledge that, as other workflows are discovered and integrated into this framework, the features and other associated properties may be extended, changed or eliminated in future versions.

5. Framework

In our study, to answer **RQ2**, we are modelling as features the instructions found in the descriptions of those workflows listed in Table 1. This set of features combines the guidelines from those descriptions and the functionality provided or supported by Git [59,60]. We selected this feature-based approach to model the Git workflows descriptions because it takes into account additional information about the features (such as their level of enforcement, variability and defining attributes) but also about their relationships (e.g., to define which features depend upon another or which combinations of features is feasible and which ones are not) which differs from traditional workflow description languages (such as WDL and CWL) that focus on the definition of data processing tasks and how these tasks are executed and in which order.

5.1. Feature-based model definitions for the Git workflows framework

A feature-based model is initially independent of the domain which it describes. However, as the detailed concepts about the subject of interest are refined, those concepts need to be translated to an abstract representation of the actual problem domain.

Applying the concepts extracted from the material collected in our survey of Git workflows, we provide the following definitions for a feature-based modelled framework in the context of our problem domain:

Definition 1. The *subject of interest* (or entity) is a Git Workflow.

Definition 2. The *features* are those guidelines or instructions that characterise the subject of interest (e.g., "the convention to follow for commit messages"). A feature may also be decomposed in *sub-features* to specify variants of the parent feature (e.g., "the direction of synchronisation between repositories can be decomposed in upstream and downstream, thus, upstream is a sub-feature of direction")

Definition 3. Our *framework* consists of all the features available to characterise Git workflows. A specific characterisation of a Git workflow (e.g., GitFlow) is an instance of such framework, containing only the relevant features that are extracted from its descriptions.

Definition 4. The features may be specified at different *levels of enforcement* in the description of a workflow, depending on how restrictive or permissive each guideline is supposed to be. The levels of enforcement for our framework followed best practices for indicating requirements levels,[3] similar to the approach used in the definition of the Git Common Flow workflow [18]. For this flow, a set of keywords (MUST, REQUIRED, SHOULD, etc.) are associated with an enforcement level. When a keyword of this type cannot be associated with a guideline then we assume it is a recommendation, as long as the guideline is part of the workflow description. In our framework, the enforcement levels are:

(i) *required features* (REQ) that must be followed if the workflow is to be complied with (e.g., "To create a new release, you MUST create a release tag"). Associated keywords: 'MUST', 'SHALL', 'REQUIRED', 'NECESSARY';
(ii) *prohibited features* (PRO) that must not be followed if the workflow is to be complied with (e.g., "You MUST NOT force push to the master branch"). Associated keywords: 'MUST NOT' or 'SHALL NOT';
(iii) *recommended features* (REC) that give the preferred way to carry out some part of the workflow, though other options are also considered compliant; if present in a group, they may be considered as a conjunction (AND), a disjunction (OR) or an exclusive disjunction (XOR) of alternatives (e.g., "It is RECOMMENDED you use "git add -i" OR "git add -p" to add individual changes"). Associated keywords: 'SHOULD', 'MAY', 'RECOMMENDED', 'OPTIONAL';
(iv) *allowed features* (ALL) that give acceptable and alternative ways of carrying out some part of the workflow. This is the default level if no guideline is present for a particular feature on a workflow description.

Definition 5. For the feature model the following elements are needed to build the corresponding feature diagram:

- the *root* element r. Represents the Git workflow to be characterised (e.g., GitFlow);
- the edges. The first level below r are the *categories* C in which features can be aggregated based on how they are used in a development process. These categories are the ones listed in Section 4.2 plus a category 0 (Framework setup), that includes those features needed by the framework, such as, which categories will be defined, and which roles correspond to which lines of development. Below the categories the remaining edges are features f characterising a Git workflow and can be classified as one of the enforcement levels of Definition 4: (i), (ii), (iii) or (iv). For example, "the main branch... must not be broken";
- the constraints con. These indicate a dependency between two features. The constraints can be of type *Requires* (e.g., for the feature "Tags" to be used, we need to select first the parent feature "File management") and *Excludes* (e.g., if the "Feature toggles" feature is mandatory for a workflow that does not use branching to organise work, then the "branching strategy" feature is not available for such workflow); and
- the attributes att. Describe additional details about a feature (e.g., a specific naming convention like "v*" for the release branch).

5.2. Semantics for feature-based modelling of Git workflows

Once the notions are defined in the context of our problem domain, we need a way to express those notions taking into account the features of a Git workflow and how these features are connected. The following semantics help expressing the features of a workflow, and their relations, following a tree-like structure (in the notation, square brackets indicate that an element is optional and curly brackets indicate that the element is mandatory) [61]:

- Feature notation:
  - $\{r\}$, is the root element or Git workflow to be described (e.g., GitFlow.);
  - $\{r.C\}$, where C is a child of r and represents a category grouping related features (Section 4.2), and $C \in \mathbb{Z} \cap \{0..6\}$ (e.g., GitFlow.4);
  - $\{r.C.f\}$, where f is a feature belonging to category C, where f is denoted by a lower-case letter (e.g., GitFlow.4.e);
  - $\{r.C.f\}[.sf]$, where sf is a sub-feature of f, where sf is denoted by an upper-case roman numeral (e.g., GitFlow.4.e.I);
  - $\{r.C.f\}[.sf]\{.rl_1\}$, rl_1 is a role, representing a development line (from Section 4.2), in our framework, where rl_1 is expressed as a lower-case roman numeral (e.g., GitFlow.4.e.I.iii); and
  - $\{r.C.f\}[.sf]\{.rl1\}\{.v|.rl2\}$, where v is a value (or selected option), and $rl2$ is second role. Either of these two alternatives defines the feature completely. If v then it is represented by an upper-case letter (e.g., GitFlow.4.e.I.iii.A). If $rl2$ then it is expressed as a lower-case roman numeral from Section 4.2 (e.g., GitFlow.5.b.iii.ii), and it is used for those features that associate two roles (for example, "*When creating a feature branch, always branch from an up-to-date master.*" [43]).
- Notation for enforcement levels (applicable to features and sub-features regardless of the example presented for each case):
  - $f \iff C$, f is a required feature of category C;
  - $sf \not\Leftrightarrow f$, sf is a prohibited sub-feature of feature f;
  - $f \implies C$, f is a recommended feature of category C; and
  - $sf \Rightarrow f$, sf is an allowed sub-feature of feature f.
- Attribute notation:
  - f: $\{att_1 = X_1, \ldots, att_n = X_n\}$, where att are the attributes of feature f and X are the set of values associated with those attributes.
- Notation for relations between categories and features, or between features and sub-features:
  - $(sf_1 \vee \cdots \vee sf_n \iff f) \wedge \neg(sf_1 \wedge \cdots \wedge sf_n)$, where $sf_1, \ldots, sf_n$ are alternative or *xor* sub-features of f; and
  - $f_1 \vee \cdots \vee f_n \iff C$, where $f_1, \ldots, f_n$ are *or* features of category C.
- Constraint notation:
  - $con_1 = \neg(f_1 \wedge f_2)$, where feature f_1 *excludes* feature f_2; and
  - $con_2 = (f_1 \iff f_2)$, where feature f_1 *requires* feature f_2.

In our framework, the full notation indicates a unique feature, and although each element of this notation is also a feature itself, it is only when put together that they can express the real meaning of a workflow guideline. In this notation, while the feature is in the middle, on the left side we have the context to which this feature belongs, expressed as abstract features for aggregation (i.e., the workflow that it describes and the category this feature is member of), and on the right side we have those concepts that refine the meaning of a feature (i.e., the role or roles it is associated with, and the values or selected options it takes when describing a specific Git workflow).

The terminology presented can be used to express the features of our framework, their properties and their relations, in a unified, structured and formal way that contrasts with the informal workflow documentation prevalent in the grey literature.

[3] https://tools.ietf.org/html/rfc2119.

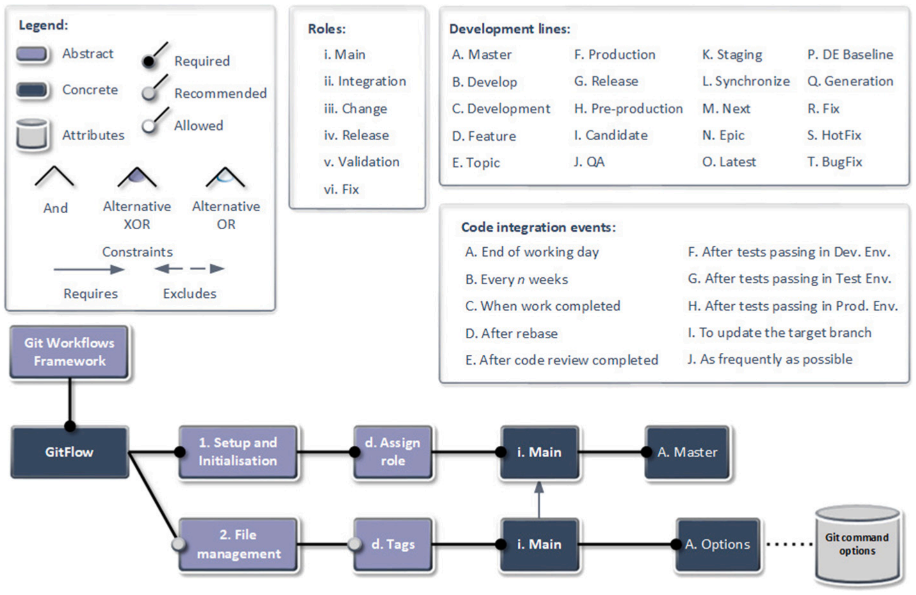


Fig. 3. Partial feature diagram of GitFlow.

5.3. Mapping a guideline to our feature-based framework

To exemplify how to map the instructions contained in the description of a Git workflow to our feature-based framework, the following guideline, taken from the original GitFlow specification [11], is converted into a feature of such workflow:

- Original guideline: "*... all of the changes should be merged back into master somehow and then tagged with a release number ...*".
- Keywords identified in the guideline: "*changes*", "*should*", "*merged*", "*master*", "*tagged*", "*release number*".
- Translation of keywords to unified terminology: "*changes*" → "*commits*", "*merged*" → "*code integrate into*", "*master*" → "*main*", "*tagged*" → "*tags*", and "*release number*" → "*version*".
- Initial interpretation: This guideline does not specify what branch should be merged into master, nor the syntax to follow for the release number, but it does specify that a tag should be used when merging into master.
- Enforcement level identified (Definition 4 in Section 5.1): "*should*" → Recommendation (REC).
- Feature construction:
  - Root element: GitFlow
  - Category (see Section 4.2): File management (2)
  - Feature (see feature example in Table 2): Tags (d)
  - Role associated (see Section 4.2): Main (i)
  - Value or option selected: Git command options (A)
  - Resulting feature (following structure from Definition 5 in Section 5.1): GitFlow.2.d.i.A (GitFlow - File management - Tags - Main - Git command options) at enforcement level: REC

This example, in feature-based modelling, can also be represented as a feature diagram, as shown in Fig. 3.

Looking closely at the mapping of the previous example, the resulting feature is following a hierarchical notation of the form: $r.C.f.rl1.v|rl2$ (following the semantic structure introduced in Section 5.2).

5.4. Defining the features

To solve the ambiguity and variability of the way the Git workflow guidelines are commonly presented, we need to define each of the features mapped in a clear and concise manner. The creation of a *dictionary of features* is a common practice in feature-based modelling [62], particularly when the domain to be modelled can be described in multiple ways depending on where the information is coming from, what is their purpose or the intended audience.

The documents containing the guidelines to follow for each Git workflow were used as the main source of information to describe the features and allow the characterisation of such workflows. Hence, in order to provide a clear definition of such features, it was important, first, to analyse the associated statements and references that can be linked to each of the resultant features. For this analysis, we applied the process described in Section 3.2.3 to extract, from the 26 workflow specifications, and from other Git resources [1,38,59,60,63,64], a list of terms concerning Git and Git workflows, to support the definition of multiple Git workflows under a consistent and structured framework.

To exemplify how the statements associated with guidelines are extracted and mapped into features, we have the following statements, that correspond to the sample feature GitFlow.2.d.i.A (GitFlow - File management - Tags - Main - Git command options), modelled in Section 5.3. Note that the workflow, from which description each statement was extracted, may not be the modelled workflow, as we are generalising a guideline to be applied indistinctly. In the following list, each statement is classified according to the types defined in Section 3.2.3:

- Git Common Flow [18]:
  - Composed statement: *"A release is just a git tag whose name is the exact release version string (e.g. "2.11.4")."*; *"A release is a git tag."*; *"To create a new release, you MUST create a git tag named as the exact version string of the release. This kind of tag MUST be referred to as a "release tag"."*; *"Creating a release is done by simply creating a git tag, typically on the master branch."*
  - Simple statement: *"You MUST NOT use a mixture of "v" prefixed and non-prefixed tags. Pick one form and stick to it."*
- GitFlow:
  - Command example and composed statement: *"Create version tags off of master."*; *"git tag -a 1.2.1"*
  - Visual statement: Fig. 1

Table 2
Feature definition examples.

Abbreviated name	Full name	Associated keywords	Related Git commands	Description
GitFlow.2.d.i.A	GitFlow - File management - Tags - Main - Git command options	tag, version, branch main, master	git tag *, git push –tags, git push –follow-tags, git describe –tags *	The participants of a development project are expected to *tag* their commits on the existing branch with role *main*
GitFlow.5.b.iii.ii.	GitFlow - Code integration strategy - Integrate into - Change - Integration	branch, merge, merged, temporary, cherry-pick, rebase, once, when, integrated back	git merge *, git cherry-pick *, git rebase *	The participants of a development project are expected to *integrate* an existing branch with role *change* into a branch with role *integration*

- ThreeFlow:
  - Command statements: *"git tag candidate-3.2.645; git push –follow-tags; git describe –tags master"*

For the extraction of the keywords and their association with Git functions we followed the steps described in Section 3.2.3, for each of the features to be defined, as shown in the example from Table 2, which partially represents two different guidelines (to simplify the examples and show different aspects of the development process) as normally these guidelines are defined by several combined features.

5.5. *List of features*

Following the steps described in Sections 3.2.2 and 3.2.3, we generated the features for our framework, up to the feature level and omitting the sub-features and values (for a list containing the full set of features please refer to Section Section 8.2 at the end of this document). This list contains the abbreviated and full naming conventions for the features in our framework, the associated keywords and Git commands, and a brief description for each feature.

Applying feature-based modelling to solve the problem of having multiple ways of describing Git workflows resulted in a list of features, that can be used to describe any of the 26 Git workflows considered for this study, using the same terminology and structure.

6. Characterising Git workflows

To address our **RQ3**, regarding the application of the feature-based framework for the characterisation of Git workflows, we created a spreadsheet (http://bit.ly/2NBAvYq) on which one of the tabs contains the list of features composing the framework. In this tab, the user of our framework is expected to mark the features that apply to a given workflow based on the analysis of its description, related to any of the workflows from the list considered for this study (as shown in Table 1).

6.1. *Instructions for workflow characterisation*

The summarised steps that must be followed, to characterise a Git workflow using our framework are:

1. Read the description of the new workflow to characterise (W) and add a pair of columns ("Enforce/Select" and "Attributes") similarly to the other workflows already characterised in the shared spreadsheet.
2. From W's description, identify the different types of repositories that play a role in the workflow, such as, "Reference" for the source repository or main code base, or "Contributor" for the contributor's working repository. To help identify the different repository types, please refer to the tab "Catalogues" in the framework's spreadsheet.
3. Determine which categories (among those described in Section 4.2) the identified guidelines belong to (e.g., repository setup or branching strategy). Identify also, which development lines (e.g., master, release or hotfix) are stipulated for W in its description. The full list of categories and development lines are given in the spreadsheet's tab "Catalogues".
4. Identify the statements indicating a guideline in the description, assigning a level of enforcement according to those in Definition 4 of Section 5.1.
5. Review the spreadsheet tabs containing foundation concepts (Dictionary, Terminology, Catalogues, Constraints) to familiarise yourself with the features to be used to characterise W.
6. Start with feature 0.a (Framework setup - Category selection), selecting the categories (1 to 6) identified in step 3 (category 0 is mandatory), under the corresponding Enforce/Select column.
7. For feature 0.b (Framework setup - Assign role), identify and assign which roles correspond to which development line(s) among those extracted from W's description. The roles identified in this feature will be used for other features and sub-features.
8. For each feature Fd identified fill the columns Enforce/Select and Attributes (if applicable) to characterise it. The attribute column can be used to specify remote repositories *R<rr>* (from those identified in step 2) or other required parameters. Dependencies between features are automatically enforced according to the relations defined on the "Constraints" tab (e.g., by selecting the category "File management" on feature 0.a.2 the rows corresponding to this category – 2.a.i to 2.f.vi – will be enabled). Following are the different types of features contained in our framework based on the information required by the feature to model a guideline:
   (a) **Simple feature/sub-feature**. Select the option that corresponds more closely to what is being described in W's description for Fd. Select the enforcement level according to Definition 4 from Section 3.2.2.
   (b) **Feature/sub-feature with attributes**. Select the option and level of enforcement for Fd. Add the attribute(s) identified for the corresponding parameter(s) as a single value or as a list.
   (c) **Feature/sub-feature with roles**. Identify to which role(s) the feature/sub-feature applies, and select the corresponding option for the relevant roles.
   (d) **Feature/sub-feature with event**. Select the appropriate value for this feature using one of the events' catalogues available at the spreadsheet's "Catalogues" tab (branch creation, branch deletion or code integration events).

To characterise a workflow using the feature-based framework, it is necessary to determine which of the features in each category are indicated in the workflow description, and with what value. For instance, if a workflow description indicates that contributors should clone the central repository for the code base directly to obtain their copy of the code, the workflow would be assigned feature '1.a.A', as either a requirement or a recommendation. If the workflow required that contributors develop through a GitHub-style fork of the main repository, then we would assign the feature '1.a.B'. If the workflow description says nothing about the on-boarding process for new contributors, then no feature value beginning '1.a' would be assigned to it. The same process would be completed for each of the features of the framework. The full characterisation of the 26 Git workflows considered in this study is included in the supplementary material for this paper.

6.2. Discussion

The process of characterising Git workflows is a combination of a methodological approach – mapping the guidelines contained in the workflow descriptions to a set of features, that represent those guidelines but using a homogeneous terminology and structure for all the workflows, and that are interconnected through hierarchical dependencies and constraints – and the judgement of a user, that translates those descriptions into the appropriate features within the framework. This process is clearly open to some level of subjectivity that is ingrained into the way those descriptions are commonly presented. This subjectivity in the descriptions is introduced by their authors in an attempt to present the guidelines and other mechanisms required for collaborative software development, and presents a problem when a user of the framework tries to interpret those subjective guidelines (e.g., "*Create stable branches using master as a starting point, and branch as late as possible.*" [43]). Therefore, the framework was designed to allow some flexibility when mapping the guidelines into features but, at the same time, applying the same structure and terminology for all the features and workflows.

Even considering the uncertainty associated with the subjective notions introduced by both the workflow creators and the users of the framework, if applied consistently, the resulting characterisations can be compared using the same notions and following the same process each time, therefore addressing **RQ3**, and allowing other mechanisms to be applied to the analysis of these workflows, as demonstrated in Sections 7.1 and 7.2.

The resulting characterisations are our proposed mechanism to compare and analyse multiple Git workflows in a systematic way, and are not intended to provide an exhaustive description of the workflows considered in this study but, instead, describe the most important elements needed to distinguish their key aspects.

7. Case studies

We envisage a range of possible uses for the proposed framework. To demonstrate this, this section presents two case studies (i.e., comparison and clustering of workflows, and the identification of families of workflows) in which the framework can be applied, to respond to our **RQ4**.

7.1. Comparing and clustering Git workflows

To compare Git workflows we performed a vectorisation of the features that characterise each of the studied Git workflows. This case study shows what results can be obtained from comparing and clustering workflows systematically, answering our question **RQ4**.

Knowing the differences and similarities between a set of Git workflows also allows their aggregation. The groups of workflows that exist are not clear without a mechanism that enables their systematic comparison. Before this study and based on the workflow descriptions, we only knew that some workflows took ideas from other popular workflows. We did not know if there are groups of workflows that focus on specific aspects of the development process (e.g., workflows that just focus on the branching strategy) or if there were other aspects that the authors of these descriptions may not be paying enough attention to (e.g., what should happen when a branch is no longer needed).

For the comparison and clustering of the Git workflows we are not requiring the enforcement levels to be exactly the same in the matched workflows (i.e., required and recommended are considered to be equivalents) because some bias may be introduced based on the guideline interpretation by the framework's user. In addition, no attributes are taken into account as this may vary depending on which project the workflow is being applied to.

7.1.1. Feature vectorisation

To create clusters from the characterised Git workflows, the user-oriented representation of the features that characterise a Git workflow was translated into a numeric vector that can be automatically processed and analysed.

The translation consists in assigning a position in the vector to each of the features of our framework. For each position in the vector we assign a numeric value that corresponds to the enforcement level, assigned during the characterisation of a workflow, using the following conversion table:

- Requirement → 2
- Recommendation → 1
- Allowed → 0
- Prohibited → -1

The values chosen serve only to differentiate the levels in the vectorised form of the workflow characterisation. They do not represent any form of ordering between levels. If no value was assigned to a feature during the characterisation (i.e., because no related guideline appeared in the workflow description) then, by default, a value of 0 (allowed) is assigned to such feature.

7.1.2. Workflows comparison

The vectorised workflow characterisations can be used to compare the 26 Git workflows through a similarity matrix, indicating how similar the vectors are once a distance measure is applied to each pair of vectors. From this comparison, we expect to see the kind of relationships – between the workflows – stated by the authors of their descriptions.

For instance, the author of the Cactus model workflow included the following statement in the blog describing this workflow [37]: "*The branching model described here is called trunk based development.*". Therefore, the expectation is that the characterisation of these two workflows should be identical or highly similar at least.

To measure the similarity between the vectors we use the Euclidean distance d (see Eq. (1)) as discussed in Section 3.4. A percentage similarity is then calculated for each workflow with all other workflows and the resulting similarity matrix is presented in Table 3. In this table, the mnemonic for each workflow, as listed in Table 1, is used instead of the full name.

The results show a high degree of similarity between all the workflows, but this is expected, as a substantial proportion of the elements in all the vectors have the same value (0 - allowed). This is because the Git workflows studied focus on specific aspects (i.e., branching and code integration strategies), and ignore other aspects of the development process.

High similarities (above 95%) were found between Cactus model and Trunk-based development, Common Flow and GitHub Flow, Fork and Fork & Merge, GitFlow and Stash Team Flow, GitHub Flow and Simple Git, and between Feature branch and Feature branch with code reviews. Contrasting these similarities with those stated in the workflow descriptions, we found all to be consistent with the authors' judgement.

On the other hand, low similarities were found between the pairs: Fork-Centralized, Fork-Fork & Merge, GitFlow-Feature branch with code reviews, and Trunk-based development-Release Flow. This is despite the fact that these workflows were associated by the authors in their descriptions.

Similarly, workflows that appeared minimally or not related based on their descriptions turned out to have a high degree of similarity, such as, GitLab Flow and Dictator & Lieutenants, and between Release Flow, GitLab Flow and Fork & Merge.

These results confirmed some of the authors' claims regarding the similarity between Git workflows, but also highlight similarities and differences that deviate from the workflow descriptions. This is an

Table 3
Similarity matrix between Git workflows (%).

	BaC	BiB	BoF	Cac	Cen	D&L	FBr	FBC	Frk	F&M	Com	GFl	GHF	GLF	GWF	IGS	Mnt	One	PLE	Rel	Sim	Skl	STF	Thr	TBD	WuF
BaC	100	80.4	75.5	65.1	73.4	76.3	73.4	68.1	78.7	67.1	60.6	62.1	66.1	70.6	67.7	72.0	69.7	68.7	65.9	70.4	66.3	62.4	63.3	65.1	72.9	65.7
BiB	80.4	100	76.9	61.4	73.1	70.4	70.4	66.3	74.8	68.5	61.7	62.6	66.7	70.8	65.9	68.7	72.2	66.1	66.1	73.4	70.6	67.9	62.4	59.8	66.7	60.6
BoF	75.6	76.9	100	68.9	82.2	74.1	78.7	73.6	80.1	70.6	70.2	61.9	76.1	70.9	69.5	71.7	76.6	68.1	64.9	85.6	80.7	74.1	62.1	62.1	77.7	67.9
Cac	65.1	61.4	68.9	100	66.3	65.1	73.6	69.1	66.1	57.4	57.9	53.7	61.9	62.3	57.1	62.9	60.9	64.2	55.4	65.7	61.7	57.8	53.5	57.1	73.6	59.0
Cen	73.4	73.1	82.2	66.3	100	80.7	80.1	73.9	89.3	69.1	66.3	60.7	72.4	70.2	66.9	73.4	75.8	68.7	65.5	75.6	77.7	71.5	59.3	66.3	75.3	62.1
D&L	76.3	70.4	74.1	65.1	80.7	100	77.4	71.5	87.3	74.1	62.3	65.7	68.1	73.9	71.5	87.6	71.1	73.1	69.5	72.2	70.4	68.5	62.9	68.7	74.3	67.7
FBr	73.4	70.4	78.7	73.6	80.1	77.4	100	93.4	81.6	64.0	69.1	59.2	76.3	70.2	68.9	72.4	72.0	70.0	60.9	78.2	76.6	71.1	58.4	62.9	82.5	64.9
FBC	68.1	66.3	73.6	69.1	73.9	71.5	93.4	100	75.1	60.6	65.5	55.7	72.0	65.7	65.7	67.3	66.9	65.1	57.2	73.6	73.6	68.5	55.0	59.0	77.4	61.4
Frk	78.7	74.8	80.1	66.1	89.3	87.3	81.6	75.1	100	68.9	64.6	60.6	70.8	70.4	70.4	79.3	77.7	70.2	66.5	72.9	73.4	68.3	59.8	71.1	77.1	62.6
F&M	67.1	68.5	70.6	57.4	69.1	74.1	64.0	60.6	68.9	100	59.8	76.1	62.6	67.5	65.5	75.6	70.0	70.2	73.4	70.6	68.5	67.5	73.4	59.2	62.6	65.9
Com	60.6	61.7	70.2	57.9	66.3	62.3	69.1	65.5	64.6	59.8	100	59.6	85.3	63.7	60.3	60.9	66.3	59.8	58.2	69.7	69.7	64.8	60.1	54.6	67.9	61.2
GFl	62.1	62.6	61.9	53.7	60.7	65.7	59.2	55.7	60.6	76.1	59.6	100	57.4	63.1	60.4	65.7	67.7	69.5	70.8	63.7	61.9	60.1	89.3	56.6	58.5	63.1
GHF	66.1	66.7	76.1	61.9	72.4	68.1	76.3	72.0	70.8	62.6	85.3	57.4	100	66.1	62.4	66.5	68.1	60.3	58.1	75.1	75.6	71.1	57.5	56.6	74.3	62.1
GLF	70.6	70.8	70.8	62.3	70.2	73.9	70.2	65.7	70.4	67.5	63.7	63.1	66.1	100	63.5	73.4	68.5	70.4	69.1	72.7	66.3	65.7	60.9	65.1	68.9	67.7
GWF	67.7	65.9	69.5	57.2	66.9	71.5	68.9	65.7	70.4	65.5	60.3	60.4	62.4	63.5	100	68.1	68.5	63.3	64.8	66.7	66.7	64.6	61.2	59.6	62.8	63.5
IGS	72.0	68.7	71.7	62.9	73.4	87.6	72.4	67.3	79.3	75.6	60.9	65.7	66.5	73.4	68.1	100	74.3	74.1	67.1	73.6	67.9	67.3	63.7	64.8	71.5	70.2
Mnt	69.7	72.2	76.6	60.9	75.8	71.1	72.0	66.9	77.7	70.0	66.3	67.7	68.1	68.5	68.5	74.3	100	71.7	69.6	76.1	70.4	66.9	65.9	62.6	66.9	66.5
One	68.7	66.1	68.1	64.2	68.7	73.1	70.0	65.1	70.2	70.2	59.8	69.5	60.3	70.4	63.3	74.1	71.7	100	68.9	71.1	64.9	62.9	65.7	60.7	68.3	69.5
PLE	65.9	66.1	64.9	55.4	65.5	69.5	60.9	57.2	66.5	73.4	58.2	70.8	58.1	69.1	64.8	67.1	69.5	68.9	100	64.9	60.7	61.2	69.3	66.5	58.7	62.6
Rel	70.4	73.4	85.6	65.7	75.6	72.2	78.2	73.6	72.9	70.6	69.7	63.7	75.1	72.7	66.7	73.6	76.1	71.1	64.9	100	80.7	78.2	61.7	60.7	75.1	68.3
Sim	66.3	70.6	80.7	61.7	77.7	70.4	76.6	73.6	73.4	68.5	69.7	61.9	75.6	66.3	66.7	67.9	70.4	64.9	60.7	80.7	100	85.3	60.4	56.8	70.8	62.6
Skl	62.4	67.9	74.1	57.8	71.5	68.5	71.1	68.5	68.3	67.5	64.8	60.1	71.1	65.7	64.6	67.3	66.9	62.9	61.2	78.2	85.3	100	57.5	56.6	66.1	61.7
STF	63.3	62.4	62.1	53.5	59.3	62.9	58.4	55.0	59.8	73.4	60.1	89.3	57.5	60.9	61.2	63.7	65.9	65.7	69.3	61.7	60.4	57.5	100	55.4	58.7	63.3
Thr	65.1	59.8	62.1	57.1	66.3	68.7	62.9	59.0	71.1	59.2	54.6	56.6	56.6	65.1	59.6	64.8	62.6	60.7	66.5	60.7	56.8	56.6	55.4	100	62.9	59.0
TBD	72.9	66.7	77.7	73.6	75.3	74.3	82.5	77.4	77.1	62.6	67.9	58.5	74.3	68.9	62.8	71.5	66.9	68.3	58.7	75.1	70.8	66.1	58.7	62.9	100	65.3
WuF	65.7	60.6	67.9	59.0	62.1	67.7	64.9	61.4	62.6	65.9	61.2	63.1	62.1	67.7	63.5	70.2	66.5	69.5	62.6	68.3	62.6	61.7	63.3	59.0	65.3	100

important result from our analysis as it can help those responsible for selecting (or creating) the workflow for a development project, by providing information about how similar two workflows are, and in which aspects of the development process lie these variations.

7.1.3. Workflows clustering

In this case study, an unsupervised machine learning technique called hierarchical clustering (Section 3.4) based on the Euclidean distances computed in the previous step was applied to the vectorised characterisations. This is a suitable clustering technique because it is applicable to numerical vectors where the number of different values is reduced (four levels of enforcement in our case), it does not need initial class labels for the data, and does not require training data (two characterisations generated for each of the 26 studied Git workflows).

For this analysis, we applied three contrasting *linkage criteria* (i.e., single, complete and Ward), that determine how the distance between the clusters is computed during the algorithm iterations, such as, the maximum distance between two observations (or feature values in our study) from each cluster (or complete linkage), the minimum distance between two observations (or single linkage), or minimising the increase in the error sum of squares – variance – between two observations (or Ward linkage) [65]. These linkages were compared to view the effect of varying such mechanisms when linking pairs of clusters (in other words, if they meet a condition they get linked, otherwise, the two clusters remain separate) in our problem domain. The result of the hierarchical clustering process is in Fig. 4.

In Fig. 4(a), the horizontal lines indicate the Euclidean distance between a Git workflow and its neighbours, and the colours indicate clusters formed following the correspondent linkage approach. In this first figure, the minimum distance (single linkage) between all the observations is used to merge pairs of clusters, thus, the algorithm starts comparing, at feature level, pairs of vectorised workflow characterisations until all the possible combinations are processed. The resulting maximum euclidean distance in the diagram was the lowest from this analysis (10.1), and results in clusters with only workflows that have a high similarity (low distance) between them. For instance, Stash Team Flow and GitFlow formed a cluster with a distance of 5.9, similarly in value to another cluster containing Fork, Dictator and Lieutenants, Centralized and IGSTK Flow, therefore, these two clusters contain the workflows most closely related according to this criterion.

In contrast, this approach prevented other similar workflows (according to their descriptions) from being clustered, as is the case of Cactus and Trunk-based development, which ended up being far from each other and inside a cluster of distance 9.2, similarly to BitBucket and BackCountry, that have a distance of 7.4 between them and therefore were not clustered together according to the single linkage criterion. Similar results occurred for a large proportion of the Git workflows, that were not paired up despite the overall high similarity, in terms of their euclidean distance, shown in Table 3. Thus we can conclude that, in this context, single linkage clusters workflows that have few identical features (i.e., with the same enforcement level) but not other workflows that have several similar features (e.g., required vs. recommended features), thus being too restrictive to model the similarities observed by just using the Euclidean distance for all the features of the Git workflows.

In Fig. 4(b), the maximum distance between all the feature values is used when building the clusters; thus, most of the workflows are paired up with another workflow. This approach follows the opposite criteria as the single linkage, thus, clustering vectors that have at least some similar values.

In this case, a maximum distance between clusters of 9.8 is obtained, and most of the similarities observed in the previous comparison analysis are confirmed, but also other pairings/clusters appeared. For instance, the Release and BOINC flows have a similarity measure of 88.46%, which in the context of all the similarity measures in Table 3 is relatively low, but they end up in the same cluster with a distance of 6.2. Another example is the large cluster containing the Feature branch, Feature branch with code reviews, Trunk-based development, Fork, Dictator & Lieutenants, and Centralized workflows, with some of them having a similarity degree between 89% and 94%. Thus, by using the complete linkage approach we can not only observe clusters – some of them with multiple workflows – that did not appear when using the single linkage, but also some of these groups include workflows that, according to the authors' descriptions, are not related.

In Fig. 4(c), the Ward approach [66] links the clusters at each iteration by minimising the variance between the observations on each pair of workflows tested. This approach is frequently used as an alternative to the other approaches of minimising or maximising the distances between the observations from two clusters that are candidates to be linked. In the five clusters that result from this approach, we find that many of the clustered workflows do have high similarity in one or more of their categories, like repository setup, branching or code integration strategies. For instance, the smaller cluster containing Feature branch, Feature branch with code reviews, Trunk-based development, and Cactus model, included two clusters with almost identical characterisations, and they all have in common their branching strategy. Another example is the large cluster containing 15 workflows with a similarity degree among their members ranging between 82% and 95%, and having multiple features in common among their branching and code integration strategies.

The Ward linkage aims to build as much clusters as possible, hence, the cluster with the highest distance between their members has a value of 9.0, an intermediate distance when compared to the other two approaches. This approach also allows the visualisation of "families" of workflows, that not only share a high similarity – in terms of their euclidean distance – between them, but they also share a number of features that may or may not be intended by their authors during their conception. For example, we noticed that for some workflows which, according to their authors, are based on others (such as GitLab Flow and GitHub Flow) it turned out that, when analysed feature by feature and clustered based on those features, their similarities are fewer than their differences, or, in a contrasting example, when a workflow was explicitly created to differentiate from another popular alternative (like Trunk-based development and Feature branch), the result is that they have more features in common than the ones on which they differ.

The three linkage approaches analysed in this Section produced different clusters of workflows, some of them corresponding with what can be inferred from the workflow descriptions, but other clusters were found as well, indicating that, when a systematical comparison between the workflows is possible, we can tell which workflows belong to the same group based on the features or categories they have in common and which ones belong to different groups, even if their description indicates otherwise.

7.1.4. Feature-level comparison

Once all the Git workflows were characterised we end up with a matrix (as the one shown in Fig. 5) that associates the workflows with the features for which each workflow provide a guidance, along with the level of enforcement required for each of the relevant guidelines (green for requirements, orange for recommendations and red for prohibitions; see Definition 4 on Section 5.1). The information collected during the characterisation allows the comparison of Git workflows at feature-level and the aggregation of those workflows based on the features they have in common.

For instance, from our analysis the Dictator & Lieutenants workflow obtained a similarity with GitLab Flow higher than expected based on their descriptions, but, when analysing what features they have in common we can see that there are multiple coincidences in the branching and code integration strategy (41.67 and 12.5% of similar features respectively), particularly for the integration and change lines of development (i.e., 4.a.ii.D, 4.a.iii.A, 4.b.ii.i, 4.b.iii.i, 5.b.ii.i, 5.b.iii.ii) whilst, in contrast, against the Git Common workflow, there is not

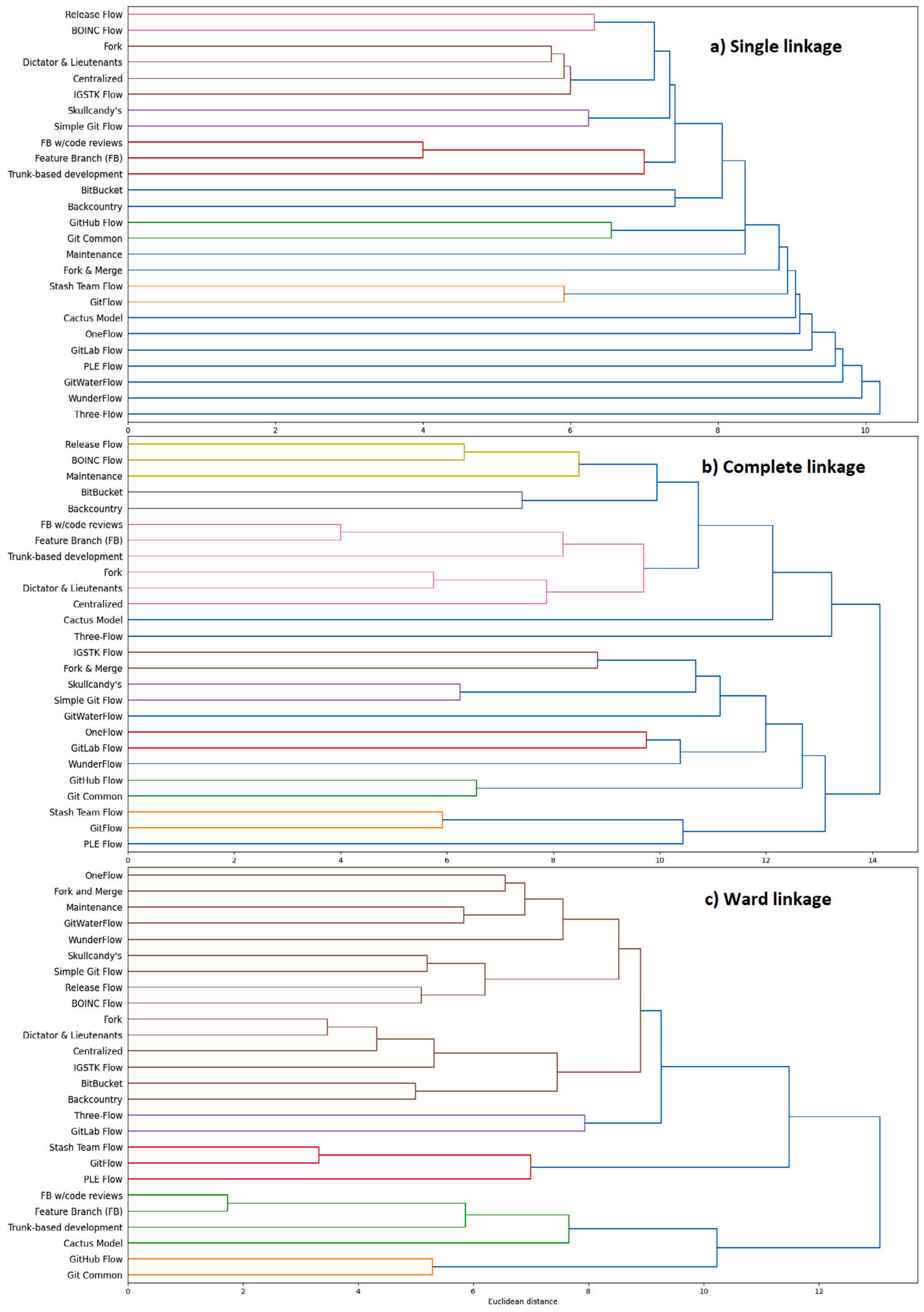


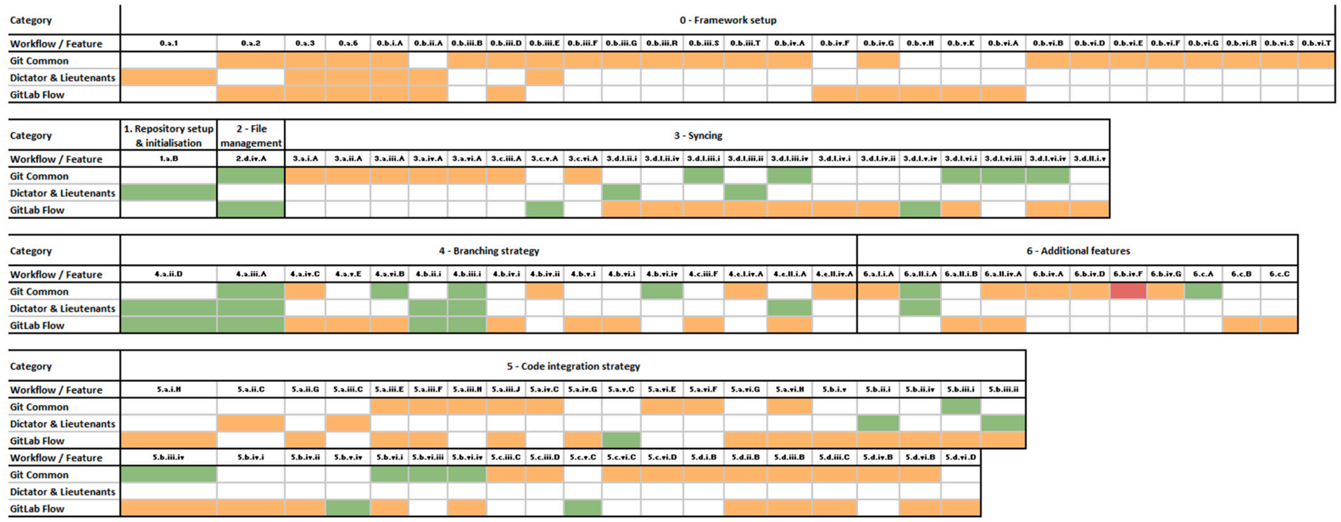
Fig. 4. Hierarchical clustering of Git workflows.

a single feature in common for categories 1, 2, 3 and 5, just two features associated with the branching strategy of the change line of development (i.e., 4.a.iii.A, 4.b.iii.i) and one additional feature for the state of development of the main line (6.a.II.i.A), which explains why the similarity between Dictator & Lieutenants and Git Common was so low.

7.1.5. Discussion

It was clear, from the initial comparison results, that the 26 Git workflows were more like each other in terms of the full set of features than a casual survey of their descriptions would suggest. Recalling that the full set of features of our framework is based not only on the Git workflows specifications but also on the functionality that Git can provide, directly (such as the Git commands for file management or code integration) or through third-party tools that enhance such functionality (i.e., feature 6.c - Supporting tools and mechanisms), it was also anticipated that many of those features may not be used by any of the workflows included in this study, but still they are important and must be included to provide a comprehensive set of functionality that can support the characterisation of other workflows that we know nothing about, and, at the same time, design the framework in a way that can accommodate other tools, mechanisms, and even new functionality from future versions of Git.

Despite the higher than expected degree of similarity between the workflows, relationships were found between certain workflow pairs, some of them expected, some not. It was only when analysing those workflows in more detail that the reasons for those similarities or differences were disclosed. This analysis revealed that, if we rely on just



Category	0 - Framework setup																											
Workflow / Feature	0.a.1	0.a.2	0.a.3	0.a.6	0.b.i.A	0.b.ii.A	0.b.iii.B	0.b.iii.D	0.b.iii.E	0.b.iii.F	0.b.iii.G	0.b.iii.R	0.b.iii.S	0.b.iii.T	0.b.iv.A	0.b.iv.F	0.b.iv.G	0.b.v.H	0.b.v.K	0.b.vi.A	0.b.vi.B	0.b.vi.D	0.b.vi.E	0.b.vi.F	0.b.vi.G	0.b.vi.R	0.b.vi.S	0.b.vi.T
Git Common		🟧	🟧	🟧	🟧		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧		🟧				🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
Dictator & Lieutenants	🟧		🟧	🟧	🟧	🟧			🟧																			
GitLab Flow		🟧	🟧	🟧	🟧	🟧		🟧								🟧	🟧	🟧	🟧	🟧								

Category	1. Repository setup & initialisation	2 - File management	3 - Syncing																			
Workflow / Feature	1.a.B	2.d.iv.A	3.a.i.A	3.a.ii.A	3.a.iii.A	3.a.iv.A	3.a.vi.A	3.c.iii.A	3.c.v.A	3.c.vi.A	3.d.I.ii.i	3.d.I.ii.iv	3.d.I.iii.i	3.d.I.iii.ii	3.d.I.iii.iv	3.d.I.iv.i	3.d.I.iv.ii	3.d.I.v.iv	3.d.I.vi.i	3.d.I.vi.iii	3.d.I.vi.iv	3.d.II.i.v
Git Common		🟩	🟧	🟧	🟧	🟧	🟧	🟧		🟧			🟩		🟩				🟩	🟩	🟩	
Dictator & Lieutenants	🟩										🟩			🟩								
GitLab Flow		🟩							🟩		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟩	🟧		🟧	🟧

Category	4 - Branching strategy																6 - Additional features										
Workflow / Feature	4.a.ii.D	4.a.iii.A	4.a.iv.C	4.a.v.E	4.a.vi.B	4.b.ii.i	4.b.iii.i	4.b.iv.i	4.b.iv.ii	4.b.v.i	4.b.vi.i	4.b.vi.iv	4.c.iii.F	4.e.I.iv.A	4.e.II.i.A	4.e.II.iv.A	6.a.I.i.A	6.a.II.i.A	6.a.II.i.B	6.a.II.iv.A	6.b.iv.A	6.b.iv.D	6.b.iv.F	6.b.iv.G	6.c.A	6.c.B	6.c.C
Git Common		🟩	🟧		🟩		🟩		🟧			🟩		🟧		🟧	🟧	🟩		🟧	🟧	🟧	🟥	🟧	🟩		
Dictator & Lieutenants	🟩	🟩				🟩	🟩								🟩			🟩									
GitLab Flow	🟩	🟩	🟧	🟧	🟧	🟩	🟩	🟧		🟧	🟧		🟧		🟧				🟧	🟧						🟧	🟧

Category	5 - Code integration strategy																			
Workflow / Feature	5.a.i.H	5.a.ii.C	5.a.ii.G	5.a.iii.C	5.a.iii.E	5.a.iii.F	5.a.iii.H	5.a.iii.J	5.a.iv.C	5.a.iv.G	5.a.v.C	5.a.vi.E	5.a.vi.F	5.a.vi.G	5.a.vi.H	5.b.i.v	5.b.ii.i	5.b.ii.iv	5.b.iii.i	5.b.iii.ii
Git Common					🟧	🟧	🟧	🟧	🟧			🟧	🟧		🟧				🟩	
Dictator & Lieutenants		🟧		🟧													🟩			🟩
GitLab Flow	🟧		🟧		🟧	🟧		🟧		🟧	🟩			🟧	🟧	🟧	🟧	🟧	🟧	🟧
Workflow / Feature	5.b.iii.iv	5.b.iv.i	5.b.iv.ii	5.b.v.iv	5.b.vi.i	5.b.vi.iii	5.b.vi.iv	5.c.iii.C	5.c.iii.D	5.c.v.C	5.c.vi.C	5.c.vi.D	5.d.i.B	5.d.ii.B	5.d.iii.B	5.d.iii.C	5.d.iv.B	5.d.vi.B	5.d.vi.D	
Git Common	🟩				🟩	🟩	🟩	🟧	🟧		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧		
Dictator & Lieutenants																				
GitLab Flow	🟧	🟧	🟧	🟩	🟧		🟧			🟩				🟧	🟧	🟧		🟧	🟧	

Fig. 5. Workflow comparison at feature-level.

the original workflow descriptions some workflows relations might be misleading, therefore the framework was able to address the problem (stated in **RQ3**) of finding unknown similarities between workflows once their systematic comparison was practically achievable.

7.2. Detecting categories shared by Git workflows

To answer **RQ4** and demonstrate how our feature-based framework can be used to identify groups or "families" of Git workflows with similar characteristics, we present the results of comparing the Git workflows at category-level (from 0 to 6 as described in Section 4.2 and in Definition 5 of Section 5.1) to determine which categories they have in common and which workflows have a higher (or lower) number of categories included in other workflows. This information might prove useful when facing the choice of several workflow candidates to follow in a new development project, to find out which ones are related (e.g., having similar branching strategies) or which ones offer contrasting approaches (e.g., regarding the use of forked or cloned repositories).

This case study compares the differences between the categories, taking into account all features that belong to each category, of different workflows, thus, when comparing two workflows characterisations (W_a and W_b), if all the features of category 1 from W_a have an equivalent enforcement level – both prohibited, or either required or recommended – in the same features but for W_b, then W_a is considered to be sharing its category 1 with W_b (regardless if W_b has other features in that category with different enforcement levels). We choose to compare categories instead of features because the comparison at feature-level has been addressed by the case study in Section 7.1.2 and also because knowing which workflows have strategies in common (branching, code integration, file management) might be more useful for decision making than knowing that two workflows enforce the same single feature.

The shared categories are then aggregated for each workflow and presented in a heat map visualisation where intense colour indicates more shared categories (up to a maximum of 7) and a light colour indicates less shared categories (down to a minimum of 0) between the Git workflows. The results in Table 4 –where the rows indicate when workflow categories are contained in other workflows and the columns indicate when a workflow contains categories from other workflows— show that the workflows that contain more categories from other workflows are: Dictator & Lieutenants from Fork (categories: 0, 1, 3, 5 and 6); Common Flow from GitHub Flow (categories: 0, 3, 4, 5 and 6); Feature branch with code reviews from Feature branch (categories: 0, 3, 4, 5 and 6); Feature branch from Feature branch with code reviews (categories: 0, 3, 4 and 6); and Git Flow from Maintenance (categories: 0, 2, 3 and 4). These similarities are expected and are well documented in those workflow descriptions, either mentioning that a workflow is based upon another or by expressing that a workflow follows a strategy that is frequently used in other workflows.

The numbers in the edges of Table 4 indicate how many times the categories are shared in total by a workflow (horizontally) or how many times categories from other workflows are included in a given workflow (vertically). For instance, the Git workflow that has its categories most frequently contained in other workflows is Fork (37), whilst the workflow that more frequently includes categories from other workflows is Stash Team Flow (24).

These results give an insight on the kind of similarities the workflows share between them and may explain why some of these workflows can be categorised as *composable* (i.e., that are normally used as a building block of other workflows and normally provide guidelines for specific features or groups of features from a single category), whilst others can be considered as *derived workflows* in the sense that they contain instructions associated with a larger number of features – and from more than one category – from other workflows.

7.2.1. Workflow families derived from shared categories

The following "families" of workflows can be assembled from the analysis of the shared categories:

1. **Singular workflows**. The following workflows have none of their categories contained in any other workflows (according to the criteria followed to build Table 4 and taking into account all the categories of the framework), indicating that they were created for a very specific and uncommon purpose or that contain a set of features that differ from those in the most popular workflows (e.g., using a different set of branching roles that have little resemblance to those recommended by two of the most popular workflows: Git Flow and GitHub Flow):

    - GitWaterFlow. This workflow follows a peculiar approach in how it propagates code changes to newer releases by employing automation tools such as bots [14].
    - PLE Flow. Workflow aimed for embedded systems projects in Product Line Engineering (PLE) that follows particular approaches for both, the Domain Engineering (DE) and Application Engineering (AE) development processes [15].

- Three-Flow. With the idea of overcoming GitFlow intricate operations around code integration, this workflow uses three branches (master, candidate and release) to organise all the work, and relies on feature toggles to propagate and test changes instead of the traditional merging mechanisms. These two characteristics make this workflow different from the others.

2. **Derived workflows**. The following workflows more frequently include the same categories (feature by feature) from other workflows, presenting some variation to address specific requirements or disadvantages of the original workflow:

   - Dictator & Lieutenants (5 categories in common with Fork). The D&L workflow relies on forking to create the hierarchical repository structure that allows multi-level control to propagate changes from the contributors to the base code.
   - Git Common (5 categories in common with GitHub Flow). This workflow includes most of the guidelines of GitHub Flow but presenting them in a more structured and formal way, including the use of enforcement levels (RFC2119) and the categorisation of the guidelines following the development process steps.
   - GitFlow (4 categories in common with Maintenance and the same branching strategy as Feature branch). One of the most popular workflows, that combines common practices from other composable workflows such as Feature branch and Maintenance.
   - Fork & Merge (4 categories in common with Fork). This workflow extends the Fork workflow, adding a code integration strategy that provides a control point to integrate work from the contributors into the code base.
   - Stash Team Flow (has the same syncing and code integration strategy categories as GitFlow but greatly differ in the branching strategy). This workflow is a variation of GitFlow, that removes the master branch to reduce the complexity of merging into multiple branches from potentially large number of feature branches.

3. **Composable workflows**. Those with a large number of features or categories included in other workflows. They can be regarded as "building-blocks" representing common practices that are frequently-used to generate new workflows:

   - Feature branch (the same branching strategy is followed by other 7 workflows). A popular approach in collaborative development to organise work on separate branches.
   - Fork (3 other workflows use the same repository setup mechanism and 8 other workflows follow the same syncing approach). This workflow mainly specifies the repository setup and syncing mechanisms to follow, and it is frequently used in other workflows mainly for organisation purposes.
   - Git Common (6 other repositories follow the same file management strategy). This workflow benefits from a clear and structured description that has allowed other workflows to replicate their guidelines regarding file management.
   - Maintenance (3 other workflows follow the same branching strategy and other 3 use the same syncing approach). This workflow mainly specifies the branching strategy for bug correction and how the resulting fixes should be managed in a distributed environment.
   - Simple Git (other 7 workflows follow the same branching strategy). The guidelines in this workflow are simplified versions of other popular workflows, that take into account frequent practices. Its simplicity makes it easier for other workflows to follow similar approaches on how to manage the branches for the development process.

7.2.2. Feature-level aggregation

Using the feature-level information from the workflows' characterisations we could also aggregate the Git workflows based on the features or group of features they have in common. For instance, from our analysis we discovered that only a few specifications, such as Centralized and Fork, describe a fully centralized approach (in which only a single line of development is used for all purposes), other specifications, like Dictator & Lieutenants, IGSTK Flow, Simple Git Flow and Three-Flow, describe workflows with a tendency towards the centralisation (between 2 and 3 different types of lines of development), whilst the rest rely on a model that uses the lines of development to organise work for different stages of the development process (i.e., development, integration, deployment, validation and fixes).

7.2.3. Discussion

The result from our exploration on the relationships between the 26 Git workflows studied was a list of which workflows are composed mostly of other workflows specifications in individual categories, and are therefore reusing proved practices that are popular in software development, or which workflows are commonly used as building blocks, or composable, as they contain guidelines on a single or few categories. These "families" of workflows can be revealed through the analysis supported by our framework, once the workflow characterisations are vectorised, and a hierarchical clustering technique is applied over them. This is our solution to **RQ4**.

This case study revealed that most workflows have more features in common than features that make them different, and variants from the more popular workflows tend to include just a few particularities that, in some cases, may not require the creation of a new named workflow. In other cases, the authors target popular workflows with the intention of debunking its claimed benefits or correcting its flaws, but use a subjective and biased approach that makes it hard to identify the actual differences or benefits from the new workflow proposed [34,47].

8. Conclusions and future work

A broad range of Git workflows have been proposed to date, espousing a variety of different approaches to collaborative development and code quality management. They offer teams a good range of options and trade-offs between the protection given to released code and the amount and type of work teams must do to achieve that. This variety, however, means appropriate selection of a workflow for a team is dependent on some team member being familiar with several contrasting workflows. While familiarity with the most well-known workflows is fairly common, an understanding of the broader range of existing workflows is much harder to acquire, since workflows cannot currently be searched for by any characteristic other than their name, and since public descriptions of some workflows are very sparse and/or incomplete.

Through a close examination of the public workflow descriptions we found a set of features that could be aggregated in a number of core workflow categories to form the basis of a common framework and vocabulary for defining workflows using a feature-based approach to associate these categories with Git functionality (directly or indirectly provided). Our experience is that workflows can be characterised through our framework in reasonable amounts of time (roughly between one and three hours of work per workflow, once the framework is understood) but that some elements of the workflow descriptions are so vaguely or informally specified that they cannot be converted into clear feature values in the framework. If the only outcome of our framework is to encourage authors of workflow specifications to use more precise language and to consider each of the core workflow elements explicitly, then the full landscape of workflows may become easier for developers and team leads to access.

Our framework can be used to characterise and compare Git workflows based on a set of features that distinguish a Git workflow from

Table 4
Workflows contained in other workflows at category-level.

	BaC	BiB	BoF	Cac	Cen	D&L	FBr	FBC	Frk	F&M	Com	GiF	GHF	GLF	GWF	IGS	Mnt	One	PLE	Rel	Sim	Skl	STF	Thr	TBD	WuF	
BaC		🟩		🟩						🟩		🟩		🟩				🟩	🟩				🟩				9
BiB	🟩																						🟩				2
BoF	🟩	🟩			🟩	🟩			🟩	🟩	🟩	🟩	🟩			🟩			🟩	🟩	🟩	🟩	🟩		🟩		21
Cac											🟩			🟩				🟩					🟩	🟩	🟩	🟩	9
Cen	🟩	🟩	🟩			🟩			🟩	🟩	🟩	🟩	🟩	🟩		🟩			🟩		🟩	🟩	🟩		🟩		19
D&L	🟩	🟩	🟩		🟩				🟩	🟩	🟩	🟩	🟩	🟩		🟩		🟩	🟩		🟩	🟩	🟩		🟩	🟩	28
FBr				🟩				🟩				🟩		🟩				🟩		🟩			🟩		🟩		14
FBC				🟩			🟩					🟩		🟩				🟩		🟩			🟩		🟩		12
Frk	🟩	🟩	🟩		🟩	🟩				🟩	🟩	🟩	🟩	🟩		🟩		🟩	🟩		🟩	🟩	🟩	🟩	🟩		37
F&M	🟩	🟩	🟩		🟩	🟩			🟩		🟩	🟩	🟩			🟩			🟩		🟩	🟩	🟩		🟩		16
Com				🟩										🟩				🟩					🟩	🟩		🟩	6
GiF										🟩							🟩		🟩				🟩				6
GHF	🟩										🟩												🟩		🟩	🟩	9
GLF				🟩							🟩							🟩					🟩	🟩		🟩	6
GWF																											0
IGS	🟩	🟩	🟩		🟩	🟩			🟩	🟩	🟩	🟩	🟩					🟩	🟩		🟩	🟩	🟩		🟩		19
Mnt										🟩		🟩						🟩	🟩				🟩				9
One				🟩							🟩			🟩									🟩	🟩		🟩	6
PLE																											0
Rel			🟩								🟩	🟩	🟩					🟩			🟩	🟩					7
Sim	🟩	🟩	🟩		🟩	🟩			🟩	🟩	🟩	🟩	🟩	🟩		🟩		🟩	🟩	🟩	🟩	🟩	🟩		🟩		28
Skl														🟩													1
STF				🟩						🟩	🟩	🟩		🟩				🟩						🟩		🟩	9
Thr																											0
TBD			🟩		🟩					🟩								🟩					🟩				5
WuF				🟩						🟩				🟩				🟩					🟩	🟩			6
	9	8	14	8	10	10	4	5	7	21	19	20	10	15	0	11	1	20	15	5	9	10	24	8	14	7	

others. Based on the analysis carried out for this work, and on the results from the scenarios where the framework was applied, we can conclude that:

- to date, there has been very limited research on formally describing, evaluating and comparing Git workflows;
- our proposed list of features can be applied to the characterisation, comparison and clustering of Git workflows to obtain meaningful results that were not evident from the available documentation of the studied workflows;
- the set of features provide a unified structure, notation and terminology that can be applied to real-world workflows, regardless of how these workflows are described;
- the workflows found do not cover all aspects considered by our framework and most of them consistently describe the same categories (i.e., branching and code integration strategy, or commit and branch naming conventions), thus, leaving room to explore other types of workflows that may exploit different aspects of the development process, such as, additional mechanisms like feature toggles and bots that may introduce substantial improvements in current development projects;
- the workflows can be aggregated in groups or "families" of workflows, as presented in Section 7.2.1 based on how they relate between them and how their guidelines are re-used when defining new workflows; and
- having more examples with guidelines not found on the studied workflows will definitely improve the scope of our research and enhance the current version of the framework and its capabilities.

By enabling the systematic comparison of Git workflows, the proposed framework supports multiple types of numerical analysis that can help a potential user in taking an objective and informed decision when facing multiple candidate workflows, that can now be assessed in terms of the similarities and differences of their characterisations, instead of relying on the advice of other users or workflow authors, that some times are prone to bias and influence from their own development environments and particular requirements. Therefore, instead of building yet more minor variants of existing workflows, workflow designers now have a way to compare candidate new and existing workflows.

8.1. Limitations of our study

An empirical comparison of 26 workflows, based exclusively on their descriptions, was contrasted with the results of the systematic comparison using the euclidean distance between the vectorised characterisations (Section 7.1.2), and the result has few points in common, such as, the similarity between workflows that are almost identical like feature branch and feature branch with code reviews. The difference between these comparisons could be caused by the subjectivity introduced in the descriptions and their characterisations, but it can also be derived from the subjectivity in the approach followed during the creation of the framework, particularly in decisions regarding the mapping of the enforcement levels and guidelines into a fixed notation, or the type and number of categories aggregating the features, or the interpretation of the guidelines by a reduced number of users. All these are known limitations of the framework, and there are factors that can be reduced as more definitions are incorporated and a deeper understanding of the development activities is assimilated into the framework, thus, this version may help to attract the attention of other workflow creators and, by combining their knowledge and experience, there can be an improvement in this mechanism to characterise and compare Git workflows.

8.2. Future work

In our future work, we will seek to refine and improve the framework, as well as exploring different ways to exploit the characterised workflows, to allow easier and more effective use of Git workflows in development projects.

To further strengthen and extend our framework, we will look at more workflow descriptions, extending the types of projects and the styles that are covered. In particular, the search performed in software development platforms for this study considered the markdown files as these files are commonly used to provide instructions and other information to users and contributors. This approach may exclude projects with idiosyncratic approaches to stating their workflows, but it made the study easier to replicate and avoid bias in establishing the framework. However, we are planning to further explore software development platforms for additional workflows, which will also allow us to explore the trend of declaring workflow rules in .md files in Git repositories.

We are also planning to validate the framework with as many of the original authors of the studied workflows as possible, to identify where our characterisations deviate from the author's intention or where the framework can be improved. Also for validation, we are working on designing and developing a tool that automatically translates a characterised workflow into a human-readable description that can then be assessed by independent parties to see if it is consistent with the original specification of such workflow.

We will delve into how to operationalise the features and implement functions to evaluate them, to automate the evaluation of a Git repository in terms of the framework's features for one or several workflows, and to identify which workflows – and to what extent – are being followed.

A tool that can assess a Git repository for its compliance with a named workflow is under development. This tool can report on the workflow steps that are not being followed by the project team, and can then be adapted to assess which workflow (or workflows) a project team is applying, even when this has not been explicitly stated in the project documentation. As this tool can allow us to compare the level of compliance of multiple projects for a given workflow, we will also be able to compare the level of compliance between different software development platforms, thus identifying the platforms that contribute better practices more, by assuming that high compliance correlates with good development practices.

In addition to the internal developers, external developers may also contribute to the projects via pull requests. It is worthwhile to investigate whether both internal and external developers follow the workflow declared for the projects or how their practices differ in respecting the workflow rules.

This in turn will enable a wider analytical study of the use of workflows in public hosted projects, such as those on GitHub.com and GitLab.com. We will be able to answer questions about the workflows that are most popular, which workflows are easy or difficult to comply with, and which variants of popular workflows are used for projects of different kinds.

Having grouped the workflows in our study into families, it may be possible to identify a representative workflow for each family, and to describe other members as variants that could be selected for particular project needs. This would form the basis of a workflow selection tool: the user ticks the elements they need for their project, and the tool suggests workflows that could best meet those needs.

Open Data

The full framework for the systematic characterisation and comparison of Git workflows can be accessed at http://bit.ly/2NBAvYq. The descriptions, for the 26 Git workflows considered for this study, has been compiled in the following Google Doc: http://bit.ly/2KOS4TP. The most recent information about the research described in this article is maintained at the following website: https://gitworkflows.cs.manchester.ac.uk/.

CRediT authorship contribution statement

Julio César Cortés Ríos: Conceptualisation, Methodology, Investigation, Writing – original draft. **Suzanne M. Embury:** Conceptualisation, Validation, Resources, Writing – review & editing, Supervision. **Sukru Eraslan:** Validation, Formal analysis, Writing – review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The work reported in this paper was supported by the Institute of Coding, which received funding from the Office for Students (OfS) and the University of Manchester, United Kingdom, and support from the Higher Education Funding Council for England (HEFCE).

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.infsof.2021.106811.

References

[1] J. Loeliger, M. McCullough, Version Control with Git: powerful Tools and Techniques for Collaborative Software Development, "O'Reilly Media, Inc.", Sebastopol, CA, USA, 2012.
[2] S. Phillips, J. Sillito, R. Walker, Branching and merging: an investigation into current version control practices, in: Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, ACM, New York, NY, USA, 2011, pp. 9–15.
[3] M. Greiler, C. Bird, M.-A. Storey, L. MacLeod, J. Czerwonka, Code Reviewing in the Trenches: Understanding Challenges, Best Practices and Tool Needs, MSR-TR-2016-27, Microsoft, 2016.
[4] K. Gadzinowski, Trunk-based development Git Flow, 2010, https://www.toptal.com/software/trunk-based-development-git-flow.
[5] L. Mezzalira, Git Flow vs Github Flow, 2014, https://lucamezzalira.com/2014/03/10/git-flow-vs-github-flow/.
[6] S. Patel, The problem with Git flow, 2020, https://about.gitlab.com/blog/2020/03/05/what-is-gitlab-flow/.
[7] M. Acher, P. Collet, P. Lahire, R. France, Comparing Approaches to Implement Feature Model Composition, in: T. Kühne, B. Selic, M.-P. Gervais, F. Terrier (Eds.), Modelling Foundations and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 3–19.
[8] K. Ram, git can facilitate greater reproducibility and increased transparency in science, Source Code Biol. Med. 8 (2013) 7, http://dx.doi.org/10.1186/1751-0473-8-7, URL https://doi.org/10.1186/1751-0473-8-7.
[9] A. Frigeri, The Use of Git in Planetary Science Research, in: European Planetary Science Congress, 2018, EPSC2018–1058.
[10] H.R. Rothstein, S. Hopewell, Grey literature, in: The Handbook of Research Synthesis and Meta-Analysis, Russell Sage Foundation, 2009.
[11] V. Driessen, A successful Git branching model, 2010, https://nvie.com/posts/a-successful-git-branching-model/.
[12] GitHub, GitHub Flow, 2010, https://guides.github.com/introduction/flow/.
[13] I. Tepavac, K. Valjevac, S. Kliba, M. Mijac, Version Control Systems, Tools and Best Practices: Case Git, in: CASE27 Conference, CASE27, Zagreb, Croatia, 2015, pp. 1–10.
[14] R.B. Rayana, S. Killian, N. Trangez, A. Calmettes, GitWaterFlow: a successful branching model and tooling, for achieving continuous delivery with multiple version branches, in: Proceedings of the 4th International Workshop on Release Engineering-RELENG 2016, ACM, New York, NY, USA, 2016, pp. 17–20.
[15] R. Hellebrand, M. Schulze, M. Becker, A branching model for variability-affected cyber-physical systems, in: 2016 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems, EITEC, IEEE, Vienna, Austria, 2016, pp. 47–52.
[16] L. Montalvillo, O. Díaz, Tuning GitHub for SPL Development: Branching Models and Repository Operations for Product Engineers, in: Proceedings of the 19th International Conference on Software Product Line, in: SPLC '15, ACM, New York, NY, USA, 2015, pp. 111–120.
[17] K. Gary, Z. Yaniv, O. Guler, K. Cleary, A. Enquobahrie, Source Code Control Workflows for Open Source Software, in: Proceedings of the International Conference on Software Engineering Research and Practice, SERP, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), Athens, Georgia, USA, 2014, p. 1.
[18] J. Myhrberg, Common Flow, 2010, https://commonflow.org/.
[19] Atlassian, Simple Git, 2010, https://www.atlassian.com/blog/archives/simple-git-workflow-is-simple.
[20] E.P. Corporation, Git workflows that work, 2010, https://www.endpoint.com/blog/2014/05/02/git-workflows-that-work.
[21] S.F. Conservancy, Git - gitworkflows Documentation, 2018, https://git-scm.com/docs/gitworkflows.
[22] V. Sharma, Comparing Git branching strategies, 2020, https://dev.to/arbitrarybytes/comparing-git-branching-strategies-dl4.
[23] Atlassian, Atlassian, comparing workflows, 2010, https://www.atlassian.com/git/tutorials/comparing-workflows.
[24] A. Lakhani, Three most popular Git workflows, 2019, http://aminlakhani.com/blog/three-most-popular-git-workflows/.
[25] B. Works, 5 Types of Git Workflows, 2019, https://buddy.works/blog/5-types-of-git-workflows.
[26] S. Hulet, Git Development Workflows: Git Flow vs. GitHub Flow, 2016, https://www.freshconsulting.com/git-development-workflows-git-flow-vs-github-flow/.
[27] Zepel, 5 Git workflows you can use to deliver better code and improve your development process, 2020, https://zepel.io/blog/5-git-workflows-to-improve-development/.
[28] P. Porto, 4 branching workflows for Git, 2010, https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf.
[29] M. Outlaw, Comparing Git workflows, 2010, https://www.codingblocks.net/podcast/comparing-git-workflows/.
[30] E. Nordfjord, Git workflow comparison, 2016, https://gist.github.com/nordfjord/fc8b4f2c0fa7e784e3de.
[31] A.J. Cruz, Types of workflows, 2010, http://drincruz.github.io/slides/git-workflow-comparison#/types-of-workflows.
[32] A. Ruka, OneFlow a git branching model and workflow, 2010, https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow.
[33] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams.
[34] R. Hilton, Three-Flow, 2010, https://www.nomachetejuggling.com/2017/04/09/a-different-branching-strategy/.
[35] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams/96-THEWORKFLOW.
[36] K. Reed, BOINC Flow, 2018, https://github.com/BOINC/boinc-policy/blob/master/Development_Documents/BOINC_Flow.md.
[37] J. Judin, A successful Git branching model considered harmful, 2010, https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful/.
[38] S. Chacon, J. Long, Git - Distributed workflows, 2014, https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows.
[39] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams/82-THEFEATUREBRANCHWORKFLOWwith_code_reviews.
[40] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams/27-FORK_WORKFLOWS.
[41] DataSift, Introducing GitFlow, 2010, https://datasift.github.io/gitflow/IntroducingGitFlow.html.
[42] Chromium, Upstream-first, 2010, https://www.chromium.org/chromium-os/chromiumos-design-docs/upstream-first.
[43] GitLab, GitLab Flow, 2010, https://docs.gitlab.com/ee/workflow/gitlab_flow.html.
[44] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams/107-THE_MAINTENANCE_WORKFLOW.
[45] M. Corp, Release Flow, 2010, https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/release-flow.
[46] M. Corp, Use Git Microsoft, 2010, https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/use-git-microsoft.
[47] S. Peters, Git with t for teams, 2013, https://es.slideshare.net/svenpeters/git-with-t-for-teams/121-THE_STASH_TEAM_FLOWat_Atlassian.
[48] P. Hammant, Trunk-based development, 2018, https://trunkbaseddevelopment.com.
[49] Wunder, WunderFlow, 2018, https://wunderflow.wunder.io.
[50] K. Pohl, G. Bockle, F.J. van Der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer Science & Business Media, 2005.
[51] K. Czarnecki, P. Grunbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool features and tough decisions: a comparison of variability modeling approaches, in: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, 2012, pp. 173–182.
[52] T. Berger, R. Rublack, D. Nair, J.M. Atlee, M. Becker, K. Czarnecki, A. Wasowski, A survey of variability modeling in industrial practice, in: Proceedings of the Seventh International Workshop on Variability Modelling of Software-Intensive Systems, 2013, pp. 1–8.

[53] S. Motavalli, S. Cheraghi, R. Shamsaasef, Feature-based modeling; An object oriented approach, Comput. Ind. Eng. 33 (1–2) (1997) 349–352.

[54] J.Y. Lee, K. Kim, A feature-based approach to extracting machining features, Comput. Aided Des. 30 (13) (1998) 1019–1035.

[55] D. Batory, Feature models, grammars, and propositional formulas, in: International Conference on Software Product Lines, Springer, 2005, pp. 7–20.

[56] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Tech. Rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[57] A.J. Viera, J.M. Garrett, et al., Understanding interobserver agreement: the kappa statistic, Fam. Med. 37 (5) (2005) 360–363.

[58] A.D. Gordon, A review of hierarchical classification, J. Roy. Statist. Soc. Ser. A (General) 150 (2) (1987) 119–137.

[59] S.F. Conservancy, Git - gitglossary Documentation, 2018, https://git-scm.com/docs/gitglossary.

[60] J. Strumpflohner, Git Explained: For Beginners, 2013, https://juristr.com/blog/2013/04/git-explained/.

[61] A. Durán, D. Benavides, S. Segura, P. Trinidad, A. Ruiz-Cortés, FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing, Softw. Syst. Model. 16 (4) (2017) 1049–1082.

[62] K. Lee, K.C. Kang, J. Lee, Concepts and guidelines of feature modeling for product line software engineering, in: International Conference on Software Reuse, Springer, 2002, pp. 62–77.

[63] Atlassian, Definitions for basic Git terminology, 2020, https://www.atlassian.com/git/glossary/terminology.

[64] E. K., Git Terms: Explained, 2019, https://linuxacademy.com/blog/linux/git-terms-explained/.

[65] B.S. Everitt, S. Landau, M. Leese, D. Stahl, Cluster Analysis, fifth ed, John Wiley, 2011.

[66] J.H. Ward Jr., Hierarchical grouping to optimize an objective function, J. Amer. Statist. Assoc. 58 (301) (1963) 236–244.