Improving and Evaluating the Git Extension Prototype "Git with Features"

Ho Thien Kim Nguyen

Abgabedatum – September 1 2025.
Software Engineering Faculty.

Erstprüfer:	Prof. Dr. Thorsten Berger
Zweitprüfer:	Prof. Dr. Daniel Strüber

Contents

Affidavit ii
Acronyms v

1 Introduction 1
1.1 Context 2
1.2 Problem 2
1.3 Research Questions 2
1.4 Approach 3
1.5 Contribution 3

2 Related Work and Technical Background 4
2.1 Automated Feature Location 4
2.2 Pro-Active Annotations 5
2.2.1 Embedded Feature Annotations 5
2.2.2 FeatRacer 5
2.3 Integration Concept of *Git with Features* 6
2.3.1 Storing Feature Information 6
2.3.2 Extending Git with Custom CLI Commands 7

3 Methodology 8
3.1 Enhancing the Git Extension Prototype 8
3.2 Experiment Design 8
3.2.1 Hypotheses and Variables 9
3.2.2 Materials and Tasks 9
3.2.3 Participants and Experiment Execution 11
3.2.4 Analysis 11
3.2.5 Mitigation of Unexpected Tool Behavior 12

4 Results for the Prototype Improvement 13
4.1 Analyzing User Feedback 13
4.2 Issues Derived From the Feedback 14
4.3 Concrete Changes to the Research Prototype 15
4.3.1 New Command Names 15

4.3.2 Manual Page and Web Documentation 15
4.3.3 `Git Feature Blame` Implementation 16

5 Results for the Experiment 17
5.1 Task Correctness 17
5.2 Completion Time 19
5.3 User Preference 20
5.4 Open Questions 23

6 Discussion 26
6.1 Answering RQ1 26
6.2 Answering RQ2 27
6.3 Threats to Validity 27

7 Conclusion 29
7.1 Thesis Outcome 29
7.2 Current Limitations 29
7.3 Future Work 30

List of Figures 31

List of Tables 32

List of Listings 33

Bibliography 34

Acronyms

AFL	Automated Feature Location
CLI	Command Line Interface
FDD	Feature-Driven Development
IDE	Integrated Development Environment
IQR	Interquartile Range
OS	Operating System
SPLE	Software Product Line Engineering
SUS	System Usability Score
XP	eXtreme Programming

1 Introduction

Software development has evolved into a highly collaborative and dynamic process. As a result, the complexity of software systems has increased significantly, leading to the emergence of new concepts, methodologies, and tools that support software developers in their work. One such concept is the notion of *features*. Features typically describe the functionality of software systems [3]. They help to keep an overview understanding of complex systems and provide a common language for different stakeholders (from developers to domain and business experts). For instance, agile methods like SCRUM, eXtreme Programming (XP) and Feature-Driven Development (FDD) organize work around features [11]. Furthermore, features help describe and distinguish software variants in software product lines (SPLs). In addition, developers often label their commits with the feature they are working on.

Many software-engineering activities are centered around features [8], such as extending or removing a feature, propagating a feature across variants, or consolidating cloned features. In order to use features to manage and evolve systems, you have to know the location of them. However, maintaining and recovering feature locations can be very time-consuming. In fact, much of a developer's time is spent searching for the feature locations [4, 10, 13].

When we hear the term "collaborative software development", we cannot imagine it without the use of version control systems. Git is the most popular free and open source distributed version control system that enables teams to manage codebases and track changes [12]. Git allows developers to have multiple local branches that can be entirely independent of each other. Developers can create new branches for each new feature they are working on, switch back and forth between them, and then delete each branch when that feature gets merged into the main line. This lays the groundwork for a feature-based workflow. However, as mentioned before, challenges in feature traceability remain. When software systems grow historically, their complexity increases, making it harder to maintain or integrate new features.

1.1 Context

Standard Git workflows do not natively support feature-specific tracking, making it difficult to isolate, maintain, and optimize features in large projects. Developers often spend excessive time manually identifying which files correspond to specific features, keeping track of them, and writing appropriate commits. The lack of streamlined feature management can slow productivity, increase development errors and complicate maintenance tasks. In order to address these challenges, T. Röthemeyer created the first prototype of the Git extension *Git with Features* as part of her Master's thesis [9]. This extension introduces abstractions to better manage feature-specific changes and dependencies in the context of feature-oriented software development.

1.2 Problem

The previous sections described the need for feature traceability support, and that the first step to address this was the creation of the *Git with Features* prototype. The next step is to determine the effectiveness and usability of this tool. Therefore, this thesis will take the next step to further analyze what this tool is still lacking, optimize it accordingly, and conduct a user study for empirical evaluation.

1.3 Research Questions

To help solve the problem, we derive the following two research questions:

RQ1: How efficient is the use of *Git with Features* in comparison to traditional Git for feature-oriented tasks?
This first research question aims at quantitatively comparing the two tools (standard Git vs *Git with Features*). It investigates whether the new tool leads to faster task completion and fewer errors.

RQ2: How do users perceive the usability and integration of *Git with Features* compared to traditional Git?
The second research question examines the subjective experience of users. It investigates whether users prefer *Git with Features* over using plain Git when performing feature-oriented tasks.

1.4 Approach

This bachelor's thesis has two goals: (1) To provide an improved version of the *Git with Features* prototype, and (2) to evaluate its usefulness by addressing the research questions outlined above.
In order to add optimizations to the prototype, we first need to identify the strengths and weaknesses of the tool. This step involves analyzing the user feedback from the initial prototype iteration. Based on these insights, corresponding development issues will be created and implemented to the prototype.
For the evaluation phase, we will design an experiment aimed at assessing the tool's effectiveness and users' perception of it. This will involve conducting a user study to collect empirical data regarding the tool's practical utility and its impact on feature-oriented tasks. Finally, we will analyze the results and user feedback to outline potential future improvements and directions for further optimization of the tool.

1.5 Contribution

This bachelor's thesis is expected to deliver an updated version of the *Git with Features* prototype as well as the respective documentation explaining how to use the tool. More importantly, it will provide empirical data from user tests that further informs potential optimizations for future iterative development. On a broader scope, this thesis will contribute to the research of supporting feature-specific tracking in Git workflows.

2 Related Work and Technical Background

As mentioned in the introduction, features help developers to communicate and implement specific program functionalities. For example, the management of features is essential in software product line engineering (SPLE), where features are described as units of variability [2, 6] – also referred to as optional features.
Previous research has proposed various feature location techniques in order to solve this problem. This section aims to give an overview of different strategies and concepts for feature traceability, and also explain the integration concept behind *Git with Features*.

2.1 Automated Feature Location

Given a feature description, the goal of automated feature location (AFL) techniques is to automatically identify the relevant code elements, such as functions, classes, or files, that implement it. AFL techniques can be broadly categorized into static and dynamic approaches [10]. Static techniques analyze the source code without executing it. One of the earliest static approaches, proposed by Chen et al. [5], uses program dependence analysis (PDA) to represent the program as a graph. This graph is traversed from an initial element (e.g., a function or global variable) to identify potentially relevant nodes.
In contrast, dynamic techniques rely on program execution. One of the earliest dynamic approaches was proposed by Wilde et al. [14]. The main idea is to collect runtime information and record which part of the code runs when a specific feature is exercised, and compare that to the runtime information when the feature of interest is inactive.
Over the years, many efforts and proposals have been made to refine and close the weaknesses of these techniques. For instance, both static and dynamic techniques can be further categorized based on the type of output they produce: plain output, which provides an unsorted list of software artifacts relevant to the feature, and guided output, which ranks its elements by relevance [10].

2.2 Pro-Active Annotations

Even though many automated feature location techniques have been proposed, Schwarz et al. [11] argue that they require project specific setup effort and often yield too many false positives to be useful in practice. Instead, they advocate to manually record locations early in the development stage using a lightweight annotation system.

2.2.1 Embedded Feature Annotations

Schwarz et al. [11] identified two main strategies for documenting features explicitly: (1) documenting feature information externally to the software assets, such as in feature databases, and (2) documenting feature information internally by embedding it into the software assets. They decided to implement the second strategy, which requires an annotation system with a unified concise syntax to embed the feature information into the software assets (e.g., code, folders, files). This resulted in the creation of a consolidated notation for embedded feature annotations where the syntax is programming-language-independent. Their research work also provided the tool FAXE (Feature Annotation eXtraction Engine) – a library for parsing and retrieving annotation [11]. Another tool that exploits these embedded annotations is FLOrIDA (Feature LOcatIon DAshboard) [1], which can extract and process feature annotations as well as visualize them by presenting views on different levels of abstractions.

2.2.2 FeatRacer

While embedded feature location produces less errors than automated feature location techniques, recording features manually can be easily forgotten by developers, and is also costly when software evolves and recordings need to be updated. Mukelabai et al. [7] presented FeatRacer, which combines active feature recording with automated feature location to overcome the limitations of both approaches. FeatRacer allows developers to continuously record features during development while utilizing a machine-learning-based recommender system to suggest potentially forgotten feature annotations. Specifically, FeatRacer learns from the commit history of a project to learn the specific use and characteristics of features and provide better recommendations. Compared to the traditional techniques, FeatRacer shows a higher average precision and is proven to be effective for small datasets.

2.3 Integration Concept of *Git with Features*

These previous works all highlight the importance of feature traceability in modern software development. They demonstrate that developing techniques and workflows to address common problems in feature-oriented software development remain an ongoing research challenge. From the two approaches described above, the integration concept of T. Röthemeyer's *Git with Features* is very similar to the idea of pro-active annotations, as it embeds feature traceability directly into the version control process.

2.3.1 Storing Feature Information

Whenever new feature information is created, it is pushed into and stored inside a separate meta-data branch. This branch is structured such that each feature has its own folder, and within these folders are subfolders for each commit associated with the feature, creating a bidirectional mapping between features and commits (Figure 2.1). Git reads this file structure and uses it to find associations between code changes and features.

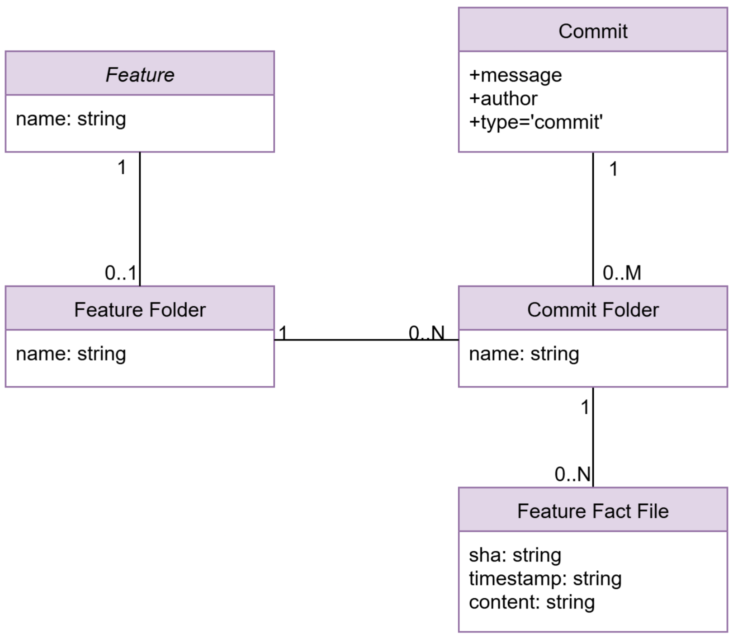


Figure 2.1: Entity-Relationship Model illustrating the association between features and commits, from [9]

2.3.2 Extending Git with Custom CLI Commands

Command	**Description**
`git feature-status`	Displays the current status of files in the working directory along with their associated features.
`git feature-add by-add`	Stages specific files or all files and associates them with the provided features.
`git feature-add from-staged`	Associates features with the currently staged files. Currently, it derives the feature list from the previously selected features for those files and is not fully implemented yet.
`git feature-commit`	Associates feature information with an existing commit.
`git feature-commits list`	Displays commits that are already associated with a feature.
`git feature-commits missing`	Displays commits that are not yet been associated with any feature.
`git feature-blame`	Shows what feature, commit and author last modified each line of a file. Optionally, specify a range of lines.
`git feature-info all`	Provides information about all features in the repository.
`git feature-info feature`	Shows a list of commits associated with a specific feature. Optionally, displays associated authors, files, branches, and update status.

Table 2.2: Git Feature Commands

There are many possibilities to extend Git, for example, forking Git and creating a separate variant, or setting custom configuration variables that control Git's behavior like Git config or Git hooks. Another option is the ability to add custom subcommands to Git. This is achieved by adding them as executable scripts in the system's PATH and referencing them with `git-<command_name>`. In the first iteration of the research prototype, each command was installed separately in the PATH. Table 2.2 shows a list of all feature commands.

3 Methodology

This chapter outlines the methodology used to enhance the research prototype and evaluate its effectiveness. First, it describes the steps made to improve the current prototype. Then, it presents the design of an experiment aimed at assessing the usability and effectiveness of the optimized tool. This includes the research variables and hypotheses, the experiment materials, and the approach used to analyze both quantitative and qualitative data.

3.1 Enhancing the Git Extension Prototype

The first objective of this thesis was to develop an enhanced version of the Git extension prototype. To inform and guide the optimization process, user feedback from the initial iteration of the prototype was analyzed, which was provided in T. Röthemeyer's master's thesis. Based on these insights, development issues were defined and sorted by priority. Additionally, incomplete commands were analyzed to determine their intended function and the order in which they should be fully implemented.

3.2 Experiment Design

Once the optimizations of the research prototype were completed, the next step was to design an experiment to gather further feedback on the *Git with Features* integration. The main purpose of the experiment was to evaluate the usability and effectiveness of the *Git with Features* tool in comparison to standard Git for feature-oriented tasks. The experiment followed a within-subjects design, meaning each participant completed tasks using both tools.
Participants had to complete eight tasks: four with standard Git and four with *Git with Features.* Three of them were feature information retrieval tasks, and one was a feature association task, respectively. To minimize learning effects, the order of tool usage varied across participants. As a result, the experiment was divided

into two phases: one in which participants used only standard Git, and another in which they used only Git Feature commands.

3.2.1 Hypotheses and Variables

In this experiment, the independent variable is the type of tool used: plain Git or *Git with Features*. Programming and professional experience were not taken into account.
The dependent variable is efficiency, which was measured quantitatively using task correctness rate and completion time. Completion time was measured in seconds, and task correctness rate was assigned by points. Task correctness was determined from the task solution answers and from the experiment recordings. Partial correctness was also noted.
For instance, if a participant only partially completed a task – such as identifying only one required file instead of two – a score of 1 was assigned to reflect partial correctness. Furthermore, we used a questionnaire to qualitatively assess user satisfaction, perceived cognitive effort, and overall tool preference.

For the two research questions stated in section 1.3, the following hypotheses were formulated, respectively:

H1: Using *Git with Features* for feature-oriented tasks is more efficient than using traditional Git.
This hypothesis was tested by examining completion time and task correctness.

H2: Users perceive *Git with Features* to be more usable for feature-oriented tasks and prefer it over plain Git.
This hypothesis was tested through a post-experiment questionnaire in which participants reflected on their experience with *Git with Features* and stated their overall tool preference.

3.2.2 Materials and Tasks

Tools. The experiment was conducted on each participant's personal computer, regardless of the OS or IDE used. The only prerequisites were that both Python and Git were installed on the system. To ensure consistency in data collection and reduce manual effort for time recording, participants were encouraged to use OBS Studio for screen recording.
Each participant was provided with the *Git with Features* package file, along with access to the Git repository in which they completed a predefined set of tasks.

The repository used in the experiment contained an implementation of a simple calculator application.
To support participants during the setup and task execution, the following resources were included: an installation guide, a list of helpful Git commands for exploring commit history, and a documentation for *Git with Features*.

Questionnaire. After the completion of the two phases, a questionnaire elicited the participants' perceptions of the respective tool. Rather than using a standardized methodology such as the System Usability Scale (SUS), we opted to create a custom set of questions. The primary reason for this decision is that SUS is a generic usability instrument that focuses on overall system interaction, which does not align closely with the specific goal and functionality of the *Git with Features* tool.
The custom questions were designed to assess user perception and satisfaction with the tool. Closed questions had a five-point Likert scale with options ranging from "strongly disagree" to "strongly agree" (e.g., *"Compared to plain Git, Git with Features helped me find relevant feature-related information more easily.", "The commands provided by Git with Features felt intuitive and logically structured.", "Git with Features integrates well with how I normally use Git."*).
Open questions aimed to be exploratory and elicit qualitative feedback (e.g., *"Were there any tasks where the tool didn't help as expected?", "What do you think needs the most improvement?", "Any suggestions for additional features or commands?"*). The final question asked which of the two tools the participant preferred for feature-oriented tasks, and invited them to explain their reasoning.

Tasks. The tasks simulated common feature-oriented development operations, such as finding information about a specific feature (e.g., commits, authors, files, branches) or adding feature annotations to commits. The following four tasks were to be solved with plain Git:

1. If you have questions about the feature subtract, whom do you ask? Write down name and e-mail address.
2. Which files were touched to implement the feature log?
3. Compare the branches feat-gui and main. Which features exist on main, but not in feat-gui?
4. Create a new feature divide and commit your changes! What commit message did you choose?

The following four tasks, which are similar to the plain Git tasks, were to be solved with Git Feature commands:

1. How many authors does the feature core have?
2. Which files were touched to implement the feature op-select?
3. Compare the branches feat-square and main. Is the main branch up-to-date regarding the feature square?
4. Create a new feature multiply, commit your changes, and associate the feature multiply with the newest commit! What commit message did you choose?

3.2.3 Participants and Experiment Execution

For the experiment, 10 computer science students were recruited who met the requirement of having basic experience with Python and Git. In order to accommodate the schedule of each participant, the experiment was designed so that they could perform it remotely. At the end of the user study, participants were asked to upload their screen recordings to the Google Drive link provided. As compensation, participants were able to enter a raffle for a voucher.
This study did not collect demographic details such as Git proficiency or level of programming skill. Given that participants meet a minimum baseline of Git and Python knowledge, further demographic distinctions were considered unnecessary, since the primary goal is to evaluate the effectiveness of the tool itself. Furthermore, since the number of participants is limited, the collection of demographic data would not have enabled any statistically significant subgroup analysis.

3.2.4 Analysis

The two metrics completion time and task correctness allowed us to test the first hypothesis.
To measure completion time, we identified the first and last edits that a user performed on a task from the screen recordings. Completion time excludes reading the task description and entering the task answer. Then, we measured the correctness of the task solutions by analyzing the screen recordings in-depth, focusing specifically on the combined usage of the commands and assigning points accordingly. For all participants and tasks, we verified that each task was completed, making a subjective assessment of whether the solution was structurally correct with only a few errors.

To answer the second hypothesis, we visualized and analyzed the questionnaire responses using stacked bar charts (Likert questions) and by inspecting the open-text responses.

3.2.5 Mitigation of Unexpected Tool Behavior

After conducting the dry-run, the following irregularities occurred:

- The `feature info` command that compares branches using the two flags `--updatable` and `--branch` did not produce the expected output, returning only an empty list. This is because the command can only compare local copies of branches, which are not available just by cloning the repository.
- For unknown reasons, the `feature status` command currently throws an error when executed.

To minimize the impact of these behaviors, we included a notice inside the task description and documentation with the instruction that remote branches be prefixed with `origin/`. Since the tasks could be solved without using this command, we also opted to not include the `feature status` command in the experiment documentation to avoid confusion with participants.

4 Results for the Prototype Improvement

In this chapter, we present the results for the prototype optimization from the methodical approach previously detailed in section 3.1.

4.1 Analyzing User Feedback

This section focuses on analyzing the user feedback collected from the initial prototype user study detailed by T. Röthemeyer [9]. This information will help determine the next steps for developing the second iteration of *Git with Features.*

Positive. The participants who favored *Git with Features* noted the following benefits: (1) Clarity in feature management, (2) ease of information retrieval, and (3) concise commands, particularly for complex projects. The first two points indicate that the tool is effective in organizing and presenting feature-related information. The third point suggests that the command structure is perceived as logical and intuitive.

Negative. Participants who were more critical of the tool mentioned three concerns: (1) Familiarity with Git, (2) interface inconsistencies, and (3) reliance on third-party tools.

- Familiarity with Git: This concern came from participants already proficient with Git, who felt no additional tool was necessary for managing feature-oriented tasks. Two possible interpretations arise: either the tool was not perceived as sufficiently novel to warrant its use alongside standard Git commands, or the experimental tasks were too simple to fully demonstrate the tool's potential advantages.
- Interface Inconsistencies: The second point suggests that the initial prototype does not yet feel like a genuine extension of Git. This raises the following

question: How can the tool be better integrated with Git so that users perceive it as a natural and seamless addition? Bridging this gap may involve refining the user interface, aligning command structures with Git conventions, or enhancing interoperability with existing Git workflows. Since this is a point where most work can be done, improving this aspect was the main focus of this thesis.

- Reliance on Third-Party Tools: Finally, some users noted the need to have Python installed as a drawback. However, this dependency cannot be easily changed, since the tool itself is implemented in Python. While it is a valid concern, it is considered out of scope for this iteration, given the technical constraints of the current implementation.

4.2 Issues Derived From the Feedback

Based on the user feedback and feasibility, we identified three main areas requiring improvement:

1. Experiment that better showcases the tool's performance in more complex scenarios
2. More intuitive command names
3. Completeness of the feature set

To highlight the tool's strength in projects with more widely scattered features, we increased the complexity of the test repository by adding more code, features, files, folders, and branches. We also revised the command structure to reduce ambiguity and align it more closely with common Git commands. Additionally, we fully implemented the `git feature blame` command, which remained incomplete in the initial version. These decisions laid the groundwork for the concrete changes detailed in the next section.

4.3 Concrete Changes to the Research Prototype

Initial Command Names	New Command Names
`git feature-status`	`git feature status`
`git feature-add by-add`	`git feature add`
`git feature-add from-staged`	`git feature add-from-staged`
`git feature-commit`	`git feature commit`
`git feature-commits list`	`git feature commits list`
`git feature-commits missing`	`git feature commits missing`
`git feature-blame`	`git feature blame`
`git feature-info all`	`git feature info-all`
`git feature-info feature`	`git feature info`

Table 4.1: Old and New Git Feature Command Names

4.3.1 New Command Names

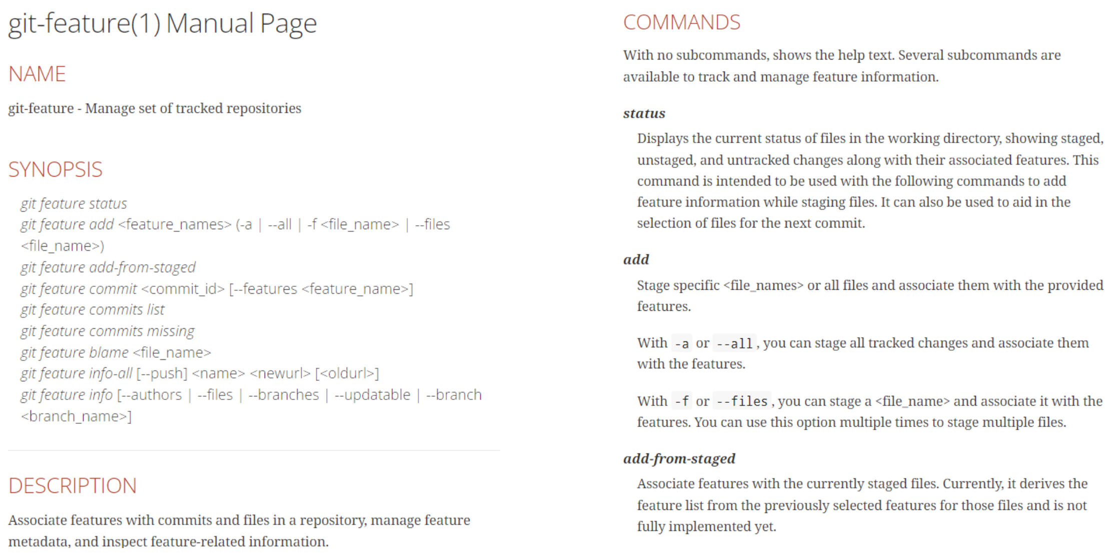
In the initial version of the tool, commands followed the following format: `git feature-<command_name>`. In the updated version, two key changes were made to improve usability and consistency with standard Git conventions:

- The dash (-) between `feature` and `<command_name>` was removed.
- Redundant wording in commands (e.g., `feature-add by-add`, `feature-info feature`) was removed.

The new command structure now mirrors the same naming pattern used in Git commands (e.g., `git status`, `git commit`, `git branch list`). This revised format aims to improve readability and make the commands less verbose.
This was achieved by assembling all git feature commands under one CLI app with `git feature` as the top-level command, and installing this app as a single executable script, instead of installing each command separately.

4.3.2 Manual Page and Web Documentation

As a byproduct of the command restructuring, the help flag now displays help text for all subcommands (except for `git feature` itself), which did not work before in the initial prototype. For the sake of completeness, we created a Manpage and a Web Documentation page for the tool, each in the likeness of standard Git documentation accessed via the help flag. However, installing these documents system-wide is not feasible without root privileges from the user, so the user has to install them manually.



git-feature(1) Manual Page

NAME

git-feature - Manage set of tracked repositories

SYNOPSIS

git feature status
git feature add <feature_names> (-a | --all | -f <file_name> | --files <file_name>)
git feature add-from-staged
git feature commit <commit_id> [--features <feature_name>]
git feature commits list
git feature commits missing
git feature blame <file_name>
git feature info-all [--push] <name> <newurl> [<oldurl>]
git feature info [--authors | --files | --branches | --updatable | --branch <branch_name>]

DESCRIPTION

Associate features with commits and files in a repository, manage feature metadata, and inspect feature-related information.

COMMANDS

With no subcommands, shows the help text. Several subcommands are available to track and manage feature information.

status

Displays the current status of files in the working directory, showing staged, unstaged, and untracked changes along with their associated features. This command is intended to be used with the following commands to add feature information while staging files. It can also be used to aid in the selection of files for the next commit.

add

Stage specific <file_names> or all files and associate them with the provided features.

With `-a` or `--all`, you can stage all tracked changes and associate them with the features.

With `-f` or `--files`, you can stage a <file_name> and associate it with the features. You can use this option multiple times to stage multiple files.

add-from-staged

Associate features with the currently staged files. Currently, it derives the feature list from the previously selected features for those files and is not fully implemented yet.

Figure 4.2: Excerpts from the web documentation

4.3.3 Git Feature Blame Implementation

The `git feature blame` command was one of the functionalities that remained only partially implemented during the first iteration. This command is designed to display the features associated within a given file, including detailed mapping of a feature to specific file lines. Below is an example usage of the command:

```
$ git feature blame src/main.py --line 3-4
op-select, core (abcd123 Jane Doe 2024-09-10 3) # code
op-select, core (abcd123 Jane Doe 2024-09-10 4) # code
```

Listing 4.3: Output Example for Git Feature Blame

In this output (Listing 4.3), each given file line of `main.py` is annotated with the corresponding feature identifiers, followed by the commit hash, author, date, and the affected line number. The format closely resembles the output of `git blame`, but extending it with the additional context of features. The intention is to make it easier for developers to answer not only *who* introduced a particular line of code, but also *why* it exists.

5 Results for the Experiment

In total, we collected around 5.5 hours of screen recordings as well as questionnaire responses from all 10 participants. This section presents the analysis results, discussing key metrics and insights from the quantitative and qualitative data collected during the user study, with the methodical approach detailed in section 3.2. Since the questions are structured such that the effort required and the information being searched for is comparable, the results for task error rate and the completion times can be measured against each other, respectively.

5.1 Task Correctness

Figure 5.1 shows the distribution of task correctness across different setups.

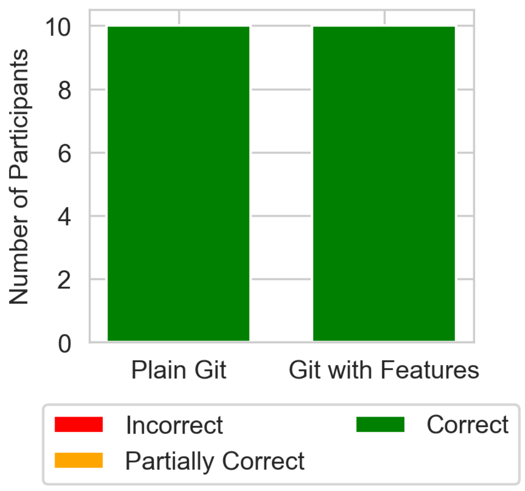
Task 1 In both the plain Git and *Git with Features* setups, all participants correctly solved the task of retrieving author information about a specific feature. In terms of correctness rates, *Git with Features* does not display an advantage over plain Git.

Task 2 Similar to Task 1, Task 2 involves retrieving file information about a specific feature. Here, the benefit of *Git with Features* is more obvious, as all participants achieved correct solutions. By contrast, the plain Git setup yields more mixed outcomes: 4 out of 10 participants solved the task incorrectly or only partially correctly.

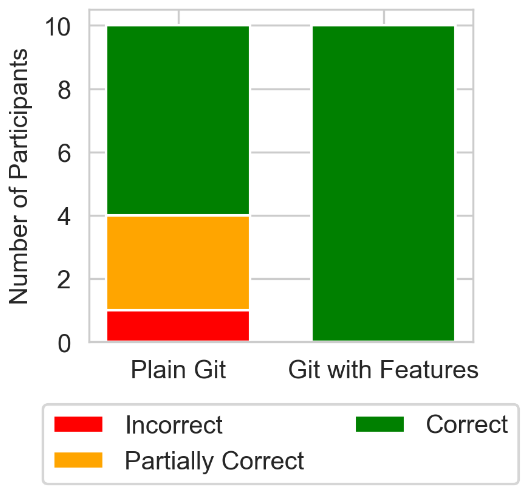
Task 3 Regarding the branch comparison task, both setups yielded mixed results, with plain Git reaching 50% correctness, while *Git with Features* reaching only 40%. This suggests that the use of the Git extension may have posed unexpected challenges.

Task 4 Figure 5.1d shows that 6 out of 10 participants correctly solved the feature association task using plain Git, whereas in the *Git with Features* setup, 9 out of 10 participants correctly solved the task. In regards to correctness rates, *Git with Features* displays an advantage over plain Git.

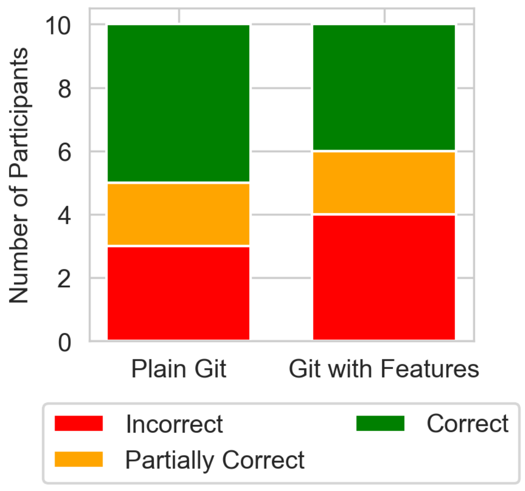
To summarize, the results indicate that for simple information retrieval tasks such as Task 1, both setups perform equally well in terms of correctness. Furthermore, *Git with Features* provides clear benefits in more complex scenarios such as file information retrieval and feature association. However, the dip in performance in the branch comparison task shows that the extension still displays some challenges.



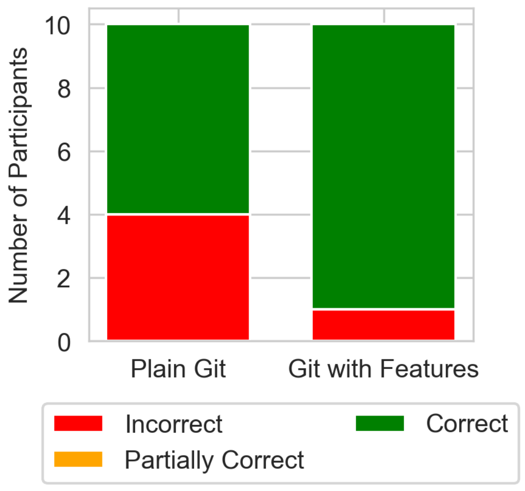
(a) Task 1



(b) Task 2



(c) Task 3



(d) Task 4

Figure 5.1: Proportion of Correctness per Task

5.2 Completion Time

Figure 5.2 shows the average time needed to complete the tasks with the different tools. Note that we are only looking at the time data of correct attempts, since "fast but wrong" results are not desirable outcomes.

Task 1 For the Plain Git setup, the median completion time is higher than that of the *Git with Features* setup. Plain Git also shows a greater variability, including an outlier at 11 minutes. This suggests that the additional tooling helped users perform the retrieval of author information more quickly.

Task 2 The difference is more noticeable here. The Plain Git setup has a higher median completion time of 2.6 minutes and a much bigger interquartile range (IQR). This indicates that users needed significantly more time to find file information when using only Plain Git. Meanwhile, *Git with Features* has a median of 0.5 minutes with a small IQR, which suggests that the use of this tool makes the file information retrieval a matter of seconds.

Task 3 The median completion time for *Git with Features* is slightly higher than that for Plain Git, but the spread is narrower with one outlier at above 8 minutes. Compared to the previous two tasks, the results do not display a benefit of *Git with Features* over Plain Git. However, the results among participants are more consistent for *Git with Features* than for Plain Git.

Task 4 Interestingly, this is the only task where Plain Git clearly outperformed *Git with Features* in regards to completion time. *Git with Features* has a significantly higher median completion time and greater spread with one outlier at above 12 minutes. This suggests that the task of feature association may have introduced some complexity or confusion.

Overall, *Git with Features* significantly outperforms Plain Git for simple feature information retrieval tasks. For Task 3, which deals with more complex information retrieval such as branch comparison, Plain Git and *Git with Features* perform similarly, with Plain Git displaying more variability among the participants. For Task 4, which involves feature association, *Git with Features* took significantly longer than Plain Git.

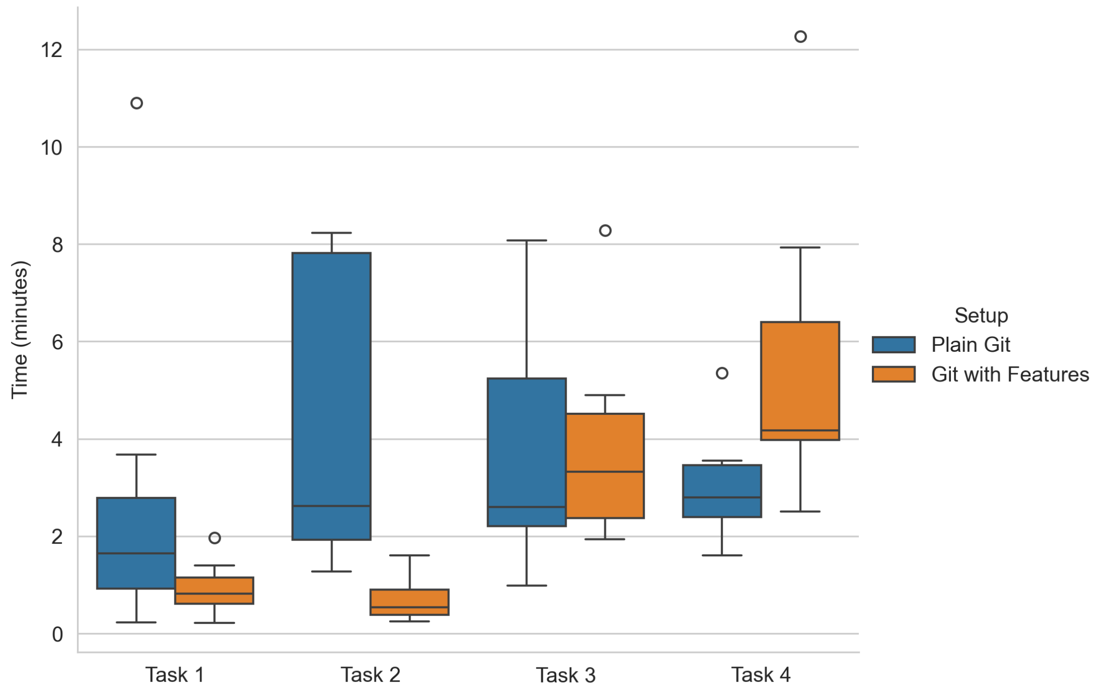


Figure 5.2: Completion Time per Task

5.3 User Preference

Figure 5.3 shows the questionnaire responses in regards to their subjective experience with using the new tool. Participants largely found the feature useful for locating relevant information, indicating strong support for its effectiveness in this area. Regarding its impact on the mental load to perform the task, the opinions are more mixed. Some users felt it reduced mental effort, but others either disagreed or were neutral, suggesting room for improvement in intuitiveness or workflow integration. Most users found the commands reasonably intuitive, though not overwhelmingly so. Some users may have found them confusing or non-standard, possibly due to unfamiliar syntax or logic. Integration into existing workflows appears to be the most controversial point. While some users appreciated the integration, others found it incompatible with their typical Git usage. This suggests either a steeper learning curve or a mismatch with expected workflows.

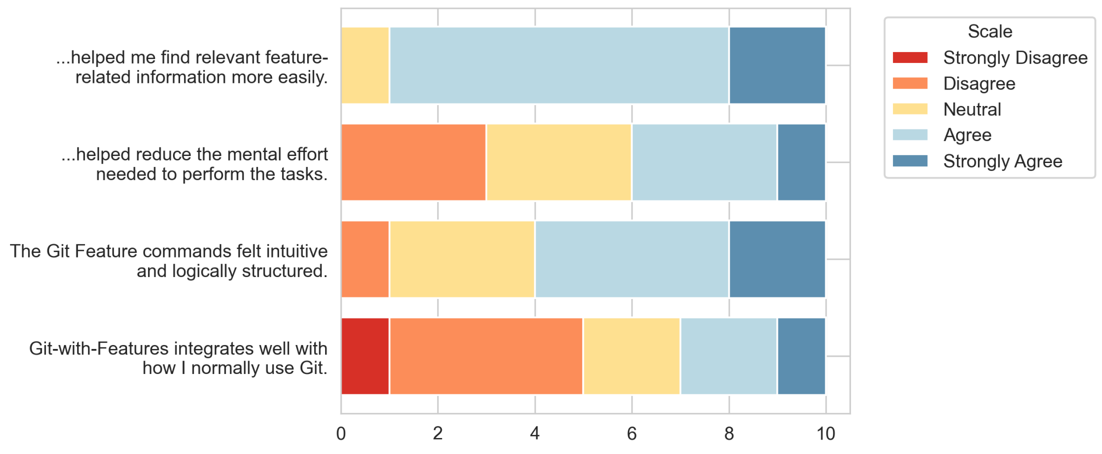


Figure 5.3: Questionnaire Responses

At the end of the questionnaire, participants were asked to state their tool preference on a scale of 1-5, where 1 indicated strong favor of traditional Git and 5 strong favor of *Git with Features*. In figure 5.4, the results are visualized. Half of the participants prefer *Git with Features*, while the rest are either neutral or prefer plain Git.

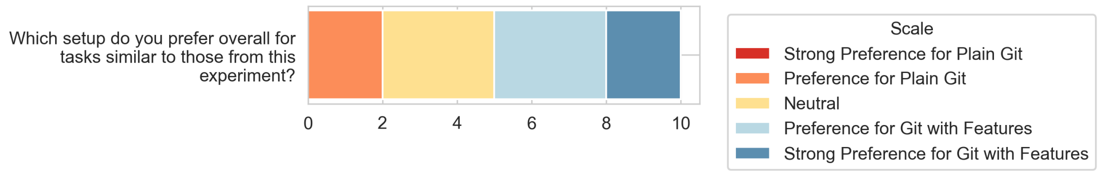


Figure 5.4: Overall Preference Responses

To gain a better understanding of the reasoning, looking into the explanations for the participant's choice can be interesting, which is listed in table 5.5. The following gives an overview of the reasoning in favor and against the tool:

- **Plain Git Preference:** Users that preferred plain Git also prefer using built-in Git functionality of IDEs instead of using Git in the CLI.
- **Neutral:** Users who had no preferences gave the reasoning that they are already familiar with existing built-in JetBrains Git UI, which they find intuitive. They also noted that the choice of tool depends on the size of the project: while the Git extension can be beneficial for smaller projects, it may become redundant in very large projects with many minor features. However,

although they are more experienced with standard Git, they remain open to using the extension when appropriate.

- **Git Extension Preference:** Users preferred the Git extension because it allowed them to work faster than with plain Git. They found that *Git with Features* made it easier to navigate between features, which they would find valuable in larger codebases and commit histories. The extension also helped them more easily understand how different parts of the code related to specific features. For the given tasks, they felt it was simpler and more efficient to complete the work using *Git with Features*. Additionally, they noted that standard Git depends heavily on well-structured commit messages and filtering, and parsing them was more cumbersome compared to the Git extension.

Preference for Plain Git:
"I dont like to work in a terminal. Normally I use in-build git functionality of PyCharm/IntelliJ",
"I dont see myself working too much with git in the cli. I also prefer Embedded Feature Annotations a little bit."

Neutral:
"I have been using the Git UI of Jetbrains and it was more intuitive.",
"Depending on the size of the project. For smaller projects like this one using Git-with-Features seems beneficial, however when projects get extremely big with a huge number of small features that are not referenced or used frequently later on it might be redundant",
"I prefer both, regarding I am more experienced in Standard Git, I am open to use the Git extension as well."

Preference for *Git with Features*:
"was faster than with git",
"Git-with-feature allows for navigating for features faster which is likely very useful in larger code bases."
"it felt easier to find how things relate to a feature"

Strong Preference for *Git with Features*:
"With the given tasks, it was easier to solve with Git with Features.",
"Standard git relies on structured commit messages and the filter options aren't that great compared to git-with-features"

Table 5.5: Overview Tool Preference Reasonings

5.4 Open Questions

To further gain a better understanding and help improve the tool in the future, I elicited qualitative feedback from three open questions. The comments are listed in the tables 5.6, 5.7 and 5.8.

Workflows. When asked about unclear or unexpected command behaviors, participants most often mentioned the branch comparison task and the commit task, since they were unsure about which command to use in certain situations and also about the output. Additionally, a comment pointed out that the tool provided little advantage over standard Git when retrieving author information, particularly in small projects.

Improvements. When asked about potential improvements, participants mentioned three main areas: (1) Simplifying and shortening commands, (2) improving installation instructions for non-Windows platforms and virtual environments, and (3) refining the workflow steps for committing and feature association. Participants also raised concerns about project scalability when a project amasses a large number of features during its development.

Suggestions. Suggestions for additional features or commands included integrating features in the git-log viewer of JetBrains IDEs, adding a new flag to the `git feature commit` command to automatically get the latest commit id and add features at the same time, and also the ability to track the feature status, e.g., "Working" and "Done".

Overall, the questionnaire responses indicate that there is room for improvement regarding the Git Feature commands and the feature association workflows, as well as providing a better installation guide for non-Windows platforms. While the core functionality was generally understood, the feedback shows that reducing command complexity and offering better integration with existing Git workflows could significantly increase adoption and efficiency for more complex tasks.

Were there any tasks where the tool didn't help as expected? Why?
"Finding authors wasn't as different here but i'd imagine the difference is greater in larger projects with more collaborators"
"when displaying the square feature and ist information, the cli didnt return any information, which made sense to me. There might not be any commits in the main branch for the feature square."
"Yes, from the first section the last task. I thought I need to find a suitable command from the list "Documentation Git Feature Commands" you provided. Maybe I should have used one, but at the end I used the standard git commands."
"'git feature add --files $filename' was not working as expected for me i.e. failed but I assume that to be an error in my installation or usage of the tool. 'git associating a commit with a feature afterwards seems to have worked on the other hand."
"comparision inbetween how branches implement feature"
"The "git feature commit" command sounds like it should commit the staged files directly into a feature"

Table 5.6: User feedback on tool workflows

What do you think needs the most improvement?
"commands can sometimes become very long, feature add command has the same name of default git add command but different functionality"
"Some changes in the wording of the documentation could have helped me but I think this issue was more specific to me personally."
"scalability of features might be a problem in git features info-all. Once the project ages many years, there could be an overview issue with so many features"
"Maybe if you show the log with all the commits added, that you can filter out the feature commits there."
"The documentation in regards to installation such as non windows platforms & virtual environments pipx etc."
"documentation got lost in the comparison taske because of it"
"Using this feature requires 2 commands for committing: (1) Regular git commit and (2) adding a commit to a feature. In my opinion, it would be much easier/better to have a single command which does both. Right now, committing is just made harder because one has to do an extra command."

Table 5.7: User feedback on areas requiring improvement

Any suggestions for additional features or commands?
"Including features in the git-log viewer of Jetbrains IDEs"
"1. adding a new feature to a commit could be easier, by using a flag –latest instead of manually getting the id with –log -1 2. Adding a feature in the git add or git commit directly would be great. Info: I have not tried if finding the difference between the main and feature branch is easier with the command line."
"Typing "git feature" feels weird, as it looks like 2 arguments and it feels like I'm using git and not an extension/add-on. To also avoid any legal complaints, I would call it "git-feature" so it is intuitive for the user that this is one argument and not really the standard git."
"Feature-status tracking? e.g. Working, Done, etc."

Table 5.8: User suggestions for additional features

6 Discussion

The objective of this thesis was to provide an updated version of the *Git with Features* prototype as well as evaluate its efficiency and usability. Now that the results of the analysis are in hand, we can address the research questions. This chapter discusses the results, including possible threats to validity that may influence the reliability of these results.

6.1 Answering RQ1

RQ1: How efficient is the use of the *Git with Features* tool in comparison to traditional Git?

The analysis shows that for simple feature information retrieval tasks, *Git with Features* achieves high efficiency compared to Plain Git, even with an ideally structured commit history. However, the tool is not yet efficient in regards to branch comparison tasks or commit-based feature association tasks. To increase the tool's efficiency for branch comparison workflows, the tool needs to yield more correct results. This can be achieved by making the feature commands more concise and also making them work with remote branches. Furthermore, to increase the tool's efficiency for feature association tasks, the respective workflow needs to be faster. This can be supported by making the usages of `git feature add` and `git feature commit` more clear and concise in the documentation, as well as introducing a new workflow that allows features to be associated directly during committing.

6.2 Answering RQ2

RQ2: How do users perceive the integration and usability of *Git with Features* compared to plain Git?

The majority of participants agreed that *Git with Features* supported them in locating relevant feature-related information more easily, suggesting an improvement in usability over plain Git. However, perceptions of the required mental effort and the intuitiveness of the new commands were more mixed, with a considerable number of participants selecting "Neutral". Additionally, with respect to integration, half of the participants felt that the tool did not fit well into their existing Git workflows. Overall, while half of the participants expressed a preference for the extension, the other half had some reservations about it. This suggests that while *Git with Features* shows promise, the current version of the prototype requires further refinement.

6.3 Threats to Validity

In this section, potential threats to the validity of the study's results are highlighted, categorized into internal, external, and construct validity. We follow the definitions of these categories as described by Wohlin et al. [15].

Internal validity. One internal threat that affects the independent variable *efficiency* without the researcher's knowledge could stem from accidental mistakes in data collection, such as misrecorded task correctness and completion times.
Another possible threat is that a learning effect may be created due to repeated testing in similar tasks but using different tools. This was mitigated by having half of the participants start with the Plain Git setup, and the other half start with *Git with Features*. However, a small learning effect could still happen when progressing from one task to the next within the same setup.

External validity. The current user study design of this thesis limits our ability to generalize the results to industrial practice. Real-world projects often have more complex branching structures, inconsistent commit messages, and varying levels of documentation. While the tool performs well in a simplified environment, it remains unknown how it would handle repositories with high technical debt or inconsistent feature tracking.
Furthermore, the experiment in its current form does not reflect a setting of collaborative software development, since each participant is solving feature-oriented

tasks on their own. This limits the insights of the tool used when multiple developers interact with the same repository at the same time.

Construct validity. The construct validity may be impacted by the evaluation metrics and tasks chosen for the user study. Currently, the tasks do not fully reflect the complexity of real-world scenarios, and they also do not elicit the use of the whole feature command set. In addition, since the experiment had to be kept small-scale, the number of feature-oriented tasks were limited. Consequently, the user study may not provide a complete picture of the tool's performance or reveal all of its potential limitations.

7 Conclusion

To conclude the bachelor's thesis, this chapter summarizes the outcomes and discusses them in a broader context. Additionally, limitations are presented alongside possible future directions from which some overlap with the directions described in T. Röthemeyer's Master's thesis [9].

7.1 Thesis Outcome

This bachelor's thesis presented an improved version of the research prototype as well as conducted an experiment that investigated the effects of *Git with Features* on feature-oriented tasks, such as feature information retrieval and commit-based feature association. The study was conducted with 10 participants, primarily computer science students with basic experience in Git and Python. The results show that efficiency with Git Feature commands can be achieved for basic feature information retrieval tasks. Moreover, the proportion of positive user response indicates that this line of research is promising.
Nonetheless, the workflows for branch comparison tasks and feature association require further refinement to ensure that the intended command functionalities become obvious to users.

7.2 Current Limitations

While the development of the *Git with Features* tool has shown promising results, the following limitations affect its current usability and scope:

1. **Incomplete Functionality of Git Feature Commands:** Due to time constraints, we could only focus on fully implementing the `feature blame` command. The user feedback, however, indicated that a few other feature commands and workflows, such as feature association, require more refinement to reliably produce their intended output.

2. **Python Dependency:** Currently, the tool cannot yet be run as a standalone program. Developers must have a recent version of Python installed, because no platform-specific distributables for Windows, macOS, or Linux are provided yet.

7.3 Future Work

After showing the results and limitations, the following points can be considered for future directions:

1. **Refining User Interface and Workflows:** In regards to the first limitation point, the focus of the next iteration should be on perfecting the functionality of the Git Feature command set. Concrete examples that need optimization can be found in sections 3.2.5 and 6.1. There are also a few interesting suggestions worth exploring, for example, displaying the development stage of each specific feature. This could give a good overview of the features in bigger projects where you can filter features by "Working" and "Done".

2. **Experimental Methodology:** Instead of just looking at how efficient the tool is during short test sessions, measurements over weeks or months can be considered to provide deeper insights and improve the reliability of the user study results. At first, users are slower because they need to learn the new commands and adapt to the new workflows. Over the long term, the initial slowness can be bridged once users get past the learning curve, so that the tool might prove to be much more efficient than it first appeared.

List of Figures

2.1 Entity-Relationship Model illustrating the association between features and commits, from [9] 6

4.2 Excerpts from the web documentation 16

5.1 Proportion of Correctness per Task 18
5.2 Completion Time per Task 20
5.3 Questionnaire Responses 21
5.4 Overall Preference Responses 21

List of Tables

2.2 Git Feature Commands 7

4.1 Old and New Git Feature Command Names 15

5.5 Overview Tool Preference Reasonings 22
5.6 User feedback on tool workflows 24
5.7 User feedback on areas requiring improvement 24
5.8 User suggestions for additional features 25

List of Listings

4.3 Output Example for Git Feature Blame 16

Bibliography

[1] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron. “FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces”. In: *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. 2017 (cit. on p. 5).

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013 (cit. on p. 4).

[3] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. “What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines”. In: *Proceedings of the 19th International Software Product Line Conference (SPLC)*. 2015 (cit. on p. 1).

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. “The concept assignment problem in program understanding”. In: *Proceedings of the 15th International Conference on Software Engineering (ICSE)*. 1993 (cit. on p. 1).

[5] K. Chen and V. Rajlich. “Case Study of Feature Location Using Dependence Graph”. In: *Proceedings of the 8th International Workshop on Program Comprehension (IWPC’00)*. IEEE, 2000, pp. 241–249 (cit. on p. 4).

[6] P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002 (cit. on p. 4).

[7] M. Mukelabai, K. Hermann, T. Berger, and J.-P. Steghöfer. “FeatRacer: Locating Features Through Assisted Traceability”. In: *IEEE Transactions on Software Engineering* 49.12 (Dec. 2023) (cit. on p. 5).

[8] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo. “Feature-oriented software evolution”. In: *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*. 2013 (cit. on p. 1).

[9] T. V. Röthemeyer. “Extending Git for Feature-Oriented Development”. M.S. thesis. NRW, Germany: Chair of Software Engineering, Ruhr-Universität Bochum, 2024 (cit. on pp. 2, 6, 13, 29).

[10] J. Rubin and M. Chechik. “A Survey of Feature Location Techniques”. In: *Domain Engineering*. 2013 (cit. on pp. 1, 4).

[11] T. Schwarz, W. Mahmood, and T. Berger. “A Common Notation and Tool Support for Embedded Feature Annotations”. In: *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B (SPLC ’20)*. New York, NY, USA: Association for Computing Machinery, 2020 (cit. on pp. 1, 5).

[12] Git SCM. *Git — Branching and Merging*. URL: `https://git-scm.com/about/branching-and-merging` (visited on 08/16/2025) (cit. on p. 1).

[13] J. Wang, X. Peng, Z. Xing, and W. Zhao. “How developers perform feature location tasks: a human-centric and process-oriented exploratory study”. In: *Journal of Software: Evolution and Process* 25.11 (2013), pp. 1193–1224 (cit. on p. 1).

[14] N. Wilde and M. C. Scully. “Software Reconnaissance: Mapping Program Features to Code”. In: *Journal of Software Maintenance: Research and Practice* 7.1 (1995), pp. 49–62 (cit. on p. 4).

[15] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012 (cit. on p. 27).