Overhead of Recording Feature Locations with Embedded Annotations

Johan Martinson[1,3][0000−0002−4097−4374], Kevin Hermann[1][0009−0004−6207−4045], and Thorsten Berger[1,2][0000−0002−2334−5838]

[1] Ruhr University Bochum, Bochum, Germany
{johan.martinson,kevin.hermann,thorsten.berger}@rub.de
[2] Chalmers | University of Gothenburg, Gothenburg, Sweden
[3] Centiro Solutions AB, Borås, Sweden

Abstract. Development processes are often organized around features. Whether to maintain or evolve features, developers frequently need to find their locations in the codebase. However, these locations are rarely recorded, which results in a substantial effort in locating features—a challenge that will intensify with the increase of AI-generated code. Many techniques, such as automatic feature location, external traceability databases, or modular feature-oriented programming, facilitate feature location, but are either imprecise, require extensive maintenance, or are hard to adopt in practice. Embedded feature annotations offer a lightweight alternative that enables developers to record feature locations directly in implementation assets as they write them. While prior work has demonstrated long-term benefits of embedded feature annotations by reducing feature location overhead, the time and workload required to create them during development remains unknown.
We conducted a controlled experiment with 15 professional developers recording feature locations in an industrial software system using embedded feature annotations. Our results show that creating embedded feature annotations entails only small overhead, requiring only 6.27% of development time with no meaningful increase in perceived workload. Although participants recognized their long-term value, they still expressed challenges in adopting them into their usual workflow—calling for more immediate benefits during development, and better tool support to reduce the effort of creating annotations.

Keywords: features · feature tracing · embedded feature annotations

1 Introduction

In software engineering, the notion of feature is typically used to categorize functionalities [14,18,28,32], to manage releases of software systems [4], and to organize development teams [34,42]. Especially agile methods, such as SCRUM, XP, or FDD use features, which encapsulate the requirements and capabilities that stakeholders seek. However, developers do not only create features, but also frequently need to change or maintain existing ones. Whether it is enhancing

functionality or addressing security vulnerabilities, quickly locating features is crucial. In fact, feature location is one of the most common activities of developers as they need to recover the implementation of features when maintaining or evolving them [17,33,50,59].

Even when development processes use features, the corresponding implementation assets, such as code, are rarely linked to them. After implementing features, their locations are usually not recorded and, therefore, easily forgotten. It is then up to the developers to recover their locations [17,21,33,50,59] when they need to maintain or evolve features—a laborious and error-prone task when done manually, especially if knowledge about them has faded. In large systems, features are often scattered over the codebase [45], interact with each other [5], and cross-cut each other's code, which challenges both developers and automated techniques. Even though many automated techniques have been presented, they often produce too many false positives and do not scale well enough for large systems [1,12,13,25,50], to be usable in practice.

Making features explicit in code supports program comprehension [31,16,54,57] and helps to quickly locate features. To this end, developers could maintain traceability links that map features to their implementation assets. However, keeping traceability links consistent in external databases [49] requires extensive effort that hinders developers from their development activities [55,56]. Such heavyweight traceability techniques have not found adoption in practice, since they require substantial time and workload overhead from developers—a scarce resource in fast-paced development environments [43]. Other methods, such as software product-line engineering, have long focused on modularizing features and making them explicit within implementation assets [4,6,7,9,11,30,47,52,51]. However, these methods are often specific to a programming language or paradigm, such as object-oriented programming. Since they rely on embedding variation points for features into implementation assets, the traceability of features is then limited to the variability of software systems, only capturing optional features, ignoring mandatory features.

Embedded feature annotations [55,56,15,27,53,36] offer a lightweight alternative for recording traceability links directly in source code. Unlike heavyweight external traceability databases [49], embedded annotations co-evolve with the code during copy, reuse, or evolutionary changes. Simulation studies [15,27] estimated long-term benefits by assuming annotation times. Comprehension studies [57,31] showed that annotations help developers comprehend systems faster, and case studies only indicated the development overhead [56]. However, we **lack hard empirical data on the actual overhead developers incur while creating annotations during coding** [58]. The critical question remains unanswered: *How much overhead, in terms of time and workload, do embedded feature annotations require from developers during development?* Without such empirical data, practitioners cannot perform informed cost-benefit analyses to assess whether the long-term benefits of traceability justify the immediate productivity impact during development sprints. In addition, *it is unclear how developers perceive benefits and challenges of using embedded feature annotations.* Although the long-term

benefits of traceability are recognized, the willingness to adopt such techniques may be influenced by short-term perceptions of their usefulness and ease of use [24]. We address this research gap by contributing a controlled experiment on the overhead of employing embedded feature annotations.

We formulated two research questions:

RQ1: What is the time and workload overhead for developers to record embedded feature traceability annotations during development? We conducted a controlled experiment with 15 professional C# developers from industry performing development tasks on an industrial freight calculation system maintained by a commercial logistics company. Participants completed tasks both with and without annotations to enable comparison. We measured annotation times through screen recordings and assessed perceived workload through post-task questionnaires using NASA TLX [23].

RQ2: What are developers' perceptions of the benefits and challenges of embedded feature annotations? We analyzed both quantitative survey responses (Likert-scale questions) and qualitative feedback to identify perceived benefits (navigation speed, traceability, reduced duplication) and challenges (learning curve, naming decisions, tool integration) reported by developers.

We show that with appropriate IDE support, recording embedded feature annotations only slightly increases total development time. In fact, the additional overhead was minimal, with only **6.27% of total development time** on average. Moreover, NASA-TLX measurements revealed **no meaningful increase in perceived workload** when using annotations. Our experiment shows that features can be effectively tracked during development without imposing substantial overhead. We contribute experiment data and analysis scripts in a replication package as an online appendix [8]. We further discuss developer perceptions, revealing that while 93% recognize long-term value, adoption willingness remains moderate (53%), suggesting that short-term benefits through visualizations and immediate feedback may be crucial to motivate sustained use [24].

2 Background

We present definitions of features and feature models, feature traceability, and embedded feature annotations.

2.1 Features and Feature Models

Feature. Many definitions of the term "feature" exist. Some define features as "an abstract representation of a characteristic or end-user-visible behavior of a software system" [4,14]. Features can distinguish the individual products within a family of related software products, a.k.a. a software product line [10,4], or increase value for a customer [48,16]. However, a feature does not only represent the behavior that a user can perceive, but more precisely, "a logical unit of behavior specified by a set of functional and non-functional requirements" [18].

Feature Model. The most common and widespread way to document features is through a feature model [28,60,37]. These models serve as compact representations of all the features within a system, capturing the essential building blocks that define various aspects, qualities, or characteristics of a software system. In addition to defining individual features, they also establish relationships (such as mandatory, optional, or alternative), and specify cross-tree constraints.

2.2 Feature Traceability

Traceability methods aid in identifying the assets related to a feature [4]. They link assets, such as source code, to a feature from a feature model and thereby support tracking features during the development process. Still, establishing traceability links takes effort. The effort of establishing traceability links between features and assets depends on how explicitly features are represented in code [44]. The more explicit a feature is, the less effort tracing it requires. Features that are purely implicit in code are the most challenging to trace, as their realization in source code is seemingly unrelated to a feature in the feature model.

Multiple techniques to establish traceability links exist, which are commonly used for feature location [17,21,33,50,59]. Traceability databases maintain traceability links in an external database that point to concrete artifacts in the software system [49]. However, traceability databases require developers to perform extensive maintenance upon changes to keep them consistent and have, therefore, not found adoption in practice [55,56,58].

2.3 Embedded Feature Annotations and Attention Investment

We advocate the use of embedded feature annotations to establish traceability links. Opposed to variability annotations—which define configuration options for feature toggling, often via preprocessor directives—embedded feature annotations record the location of every feature in the codebase. Developers convert implicit features into explicit ones by annotating software assets with feature names from a feature model while writing code. Proactively recording feature locations while the feature is still fresh in the developer's mind facilitates maintaining consistency and mitigates the costs, errors, and effort of recovering traceability links retroactively after knowledge of a feature has faded [59,27].

Figure 1 shows how embedded feature annotations manifest in a system. They establish mappings of folders, files, and code fragments to features from a feature model. Folders and files are mapped through textual annotation files, while code fragments are annotated directly in source code comments, as shown in the middle of Fig. 1. For blocks of code, in-file annotations surround the code fragment at the beginning and end, while line annotations are written at the end of the line. As such, they are lightweight and programming language-independent, and suitable to establish traceability links during development.

Embedded feature annotations offer significant advantages over external traceability databases. Since they are embedded directly within artifacts, they naturally evolve alongside the code during copy, reuse, or evolutionary changes,

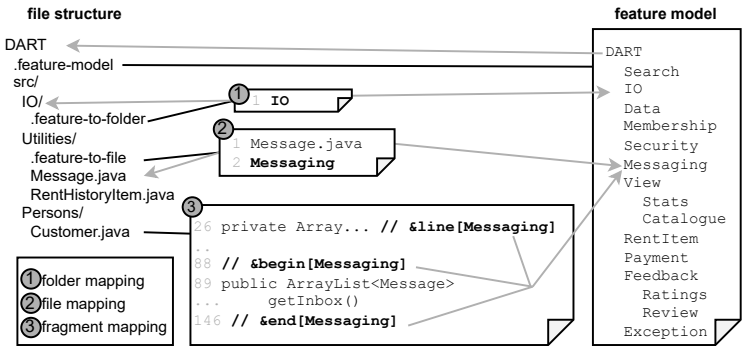

Fig. 1: Embedded feature annotation system [53]

such as refactorings, maintaining consistency without requiring extensive manual updates [15,27,53]. Berger et al. [15,27] demonstrated through simulation that, while creating embedded feature annotations requires upfront effort, the long-term benefits in reduced feature location time outweigh these costs. Similarly, Sulír et al. [57] showed that using annotations accelerates system comprehension by 34% on average. These studies establish the benefits of embedded feature annotations: once recorded, feature locations can be retrieved with minimal overhead, which saves time during maintenance and evolution.

However, *the time and workload overhead of creating feature annotations during coding has not been empirically studied*. This represents a critical knowledge gap for several reasons. First, **attention is a scarce cognitive resource** in software development [43]. Developers must invest attention in problem comprehension, code construction, and numerous other cognitive demands. Additional activities that divert attention from these tasks can feel interruptive even if they are beneficial in the long term. Second, **the perception of immediate overhead strongly influences adoption**. Even when long-term benefits are evident, developers and organizations are often reluctant to adopt practices that impose immediate productivity penalties. Finally, **overhead differs from time investment**. A task may be quick but mentally taxing (e.g., because it requires switching between two mental models), or slow but relatively automatic. Understanding both the time and workload dimensions of attention provides a more complete picture of the adoption barrier.

To minimize the overhead of creating and maintaining embedded feature annotations, developers require tool support close to their development activities [15]. Such tools can provide features like autocomplete, consistency checking, and navigation capabilities. In this study, we utilize tool support to investigate the overhead required when these productivity aids are available [36].

3 Experiment

To investigate the overhead in terms of time and workload for recording feature traceability during development we conducted a controlled experiment. We describe our experiment design by following guidelines from Jedlitschka et al. [26].

3.1 Research Goal

Our study investigates the time and perceived workload required to use embedded feature annotations during development, as well as the perceptions that professional developers have on the annotation process and its benefits. Specifically, we investigate two research questions:

RQ1: What is the overhead in terms of time and workload for developers to create embedded feature traceability annotations during development?
RQ2: How is the annotation process perceived by developers?

Our research goal is to investigate the time and workload required to create embedded feature annotations. We operationalize "overhead in terms of time and workload" through two measures: *(i)* task completion time (minutes), and *(ii)* perceived workload assessed via NASA-TLX scores (0–100 scale). Given our research goal of measuring time and workload, we measure the effect size of paired comparisons between treatment (with embedded feature annotations) and control (without embedded feature annotations) conditions.

3.2 Design

Our goal is to obtain quantitative data on the time and workload it takes to annotate code, and to explore developers' perception on recording feature locations during coding. While prior work established the benefits of annotations as self-evident (recorded feature locations save substantial feature-location effort [59]), there is no empirical measurement of the time and workload overhead during development yet. We employed a within-subject design, in which all participants completed two development tasks: one task with IDE support for embedded feature annotations (treatment condition) and one task without such support (control condition). It controls for individual differences in programming expertise, coding speed, familiarity with the subject system, and problem-solving approaches. Since each participant serves as their own control, this design minimizes variability from these individual factors, increasing statistical power and allowing us to detect effects with a smaller sample size [19].

To mitigate potential order effects (e.g., learning effects or fatigue), we counterbalanced the order in which participants encountered the treatment and control conditions. Specifically, half of the participants (Group 1, $n = 8$) completed the control task first followed by the treatment task, while the other half (Group 2, $n = 7$) completed the treatment task first followed by the control task. Task assignment was also counterbalanced. Participants who received Task 1 in the control condition received Task 2 in the treatment condition, and vice versa, as Table 1 shows. This minimizes observed differences between conditions in terms of task order, task difficulty differences, or learning effects.

Variables. The independent variable in our experiment are the embedded feature annotations along the IDE-based tool support, which are either present or absent. In the treatment condition, participants were asked to create feature annotations

	Task 1	Task 2
Group 1	Control (without annotations)	Treatment (with annotations)
Group 2	Treatment (with annotations)	Control (without annotations)

Table 1: Experiment design matrix showing task and condition assignments for counterbalancing.

during coding and had access to an IDE plugin for creating and managing embedded feature annotations. In the control condition, participants completed development tasks without access to additional tools or feature annotation support, representing standard development practice.

We measured multiple dependent variables to assess overhead and developer perceptions. **RQ1 (Overhead):** *(i)* task completion time measured in both conditions, *(ii)* time spent annotating measured in the treatment condition only, and *(iii)* perceived workload via NASA-TLX scores (0–100 scale) in both conditions. **RQ2 (Developer Perceptions):** *(iv)* Likert-scale ratings and *(v)* open-ended responses about annotation usefulness, task difficulty, and perceived effects on comprehension.

3.3 Material and Tasks

Subject System. We used an industrial freight calculation system developed and actively maintained by a commercial logistics company. The system is actively used in production to calculate shipping costs and rates for freight items and consists of 17k lines of C# code and 39 features. We selected this system because it represents a realistic development context with sufficient complexity to require feature location and comprehension efforts, while being comprehensible enough for participants to work with in an experimental session. However, we cannot publish the detailed implementation, as the system is proprietary.

Tools. We provided participants with HAnS [36], an open-source IDE plugin for JetBrains IDEs that supports creating and managing embedded feature annotations. We selected HAnS because it provides lightweight and comprehensive tool support to aid developers in recording embedded feature annotations. The plugin provides: *(i)* actions and keyboard shortcuts to create inline annotations, *(ii)* autocomplete and quick-fix actions to assist developers in writing annotations, *(iii)* consistency checking through real-time warnings when annotations reference undefined features or when feature model definitions lack usage in code, *(iv)* a feature model view that displays the hierarchical feature structure and enables navigation to annotated code locations, and *(v)* refactoring support to rename features across the codebase while maintaining consistency between annotations and the feature model. We provided participants with a demo video explaining the plugin's functionalities and the underlying annotation system.

All experiment sessions were conducted remotely, except for one participant for whom we provided a local setup. We conducted full experiment sessions, including training, control and treatment conditions, individually over Microsoft

Teams [40]. For screen recordings, we used either its built-in recording tool or integrated snipping tool, depending on participant preference.

Questionnaire. Before the experiment, we asked participants to fill out a pre-experiment questionnaire. We utilized five-point Likert-scales to assess participant demographics (current role and years of professional development) and their technical background. To collect data on the workload we used the NASA TLX on the standard 0-100 point scale [23], administered after for each condition. The NASA TLX assesses six dimensions of perceived workload: mental demand, physical demand, temporal demand, performance, effort, and frustration. Additionally, we used a post-experiment questionnaire with questions about perceived difficulty of tasks, and perceptions on embedded feature traceability annotations in the treatment condition. Finally, we posed open-ended questions to provide additional insights about the perceived benefits of embedded feature annotations, experiences with the plugin, as well as potential improvements.

Preparatory Tasks. Prior to the experiment, we asked participants to install HAnS in the JetBrains Rider IDE (an IDE for C# development) and briefed them about the subject system and the experimental procedure. We then gave participants an introduction to embedded feature annotations and their purpose.

To familiarize participants with the annotation process, we designed a structured 10-15 minute warm-up task that covered all plugin features and annotation patterns they require. The warm-up task required participants to: *(i)* add features to the feature model, *(ii)* create line-level and block-level annotations using manual typing, autocomplete, and the "surround with" feature, *(iii)* browse annotated code using the feature model viewer, *(iv)* modify annotations and use quick-fixes, and *(v)* maintain annotations during refactoring. These tasks ensured participants were trained in using the plugin before the tasks.

Tasks. Once participants familiarized themselves with the plugin, they completed two development tasks. Each task required participants to locate relevant code, understand feature boundaries, annotate the implementation with feature annotations (in the treatment condition), and make the following modifications.

Task 1: Implement Validation Logic. Participants implemented address validation logic to ensure that certain address types contain required fields (postal code or zone code) before zone matching is attempted. The task required participants to: *(i)* add a validation method that checks address types and field presence, *(ii)* add a feature flag to control the validation, *(iii)* ensure the implementation integrates correctly with existing code.

Task 2: Extract and Refactor Functionality. Participants extracted postal code normalization logic (removing spaces and hyphens) from one location in the codebase and moved it to a more appropriate class (the Address domain object) where it could be reused. The task required participants to: *(i)* locate the existing normalization logic within a feature, *(ii)* extract it into a new method on the Address class, *(iii)* update the original call site to use the new method.

3.4 Participants

We recruited 15 professional C# developers from the organization maintaining the subject system. All participants were familiar with the subject system domain and architecture, though their familiarity with the codebase varied, with some participants working with the codebase for the first time.

Participants rated their programming expertise on a five-point Likert scale and reported years of experience. They had medium to high expertise ($M = 3.73 \pm 0.88$) and substantial experience ($M = 7.2 \pm 4.3$ years). Participants had low to moderate familiarity with feature modeling and traceability concepts ($M = 2.40 \pm 0.83$), making them representative of developers encountering embedded feature annotations for the first time. Consequently, no participant had prior experience with the HAnS plugin. Participation in the study was voluntary with no advantages or disadvantages for participants. All data was collected and stored anonymously, with participants identified only by numeric codes (P01–P15). Each experimental session was limited to 90 minutes.

3.5 Analysis

We extracted quantitative data from screen recordings and qualitative data from post-task questionnaires.

Time Investment (RQ1). We measured total task completion time and annotation time by identifying start and end points of annotation activities in screen recordings. We calculated annotation time as a percentage of total development time. To measure effect size, we compared total completion times between treatment and control conditions using Cohen's d [20].

Perceived Workload (RQ1). We analyzed NASA-TLX responses across six dimensions (mental demand, physical demand, temporal demand, performance, effort, frustration) and computed overall workload scores by averaging the six dimensions (with performance inverted: reported value subtracted from 100, so higher scores indicate greater workload). We then compared overall raw NASA-TLX scores between conditions using Cohen's d.

Developer Perceptions (RQ2). We analyzed Likert-scale responses calculating means, standard deviations, and agreement percentages, and grouped similar open-ended responses to identify common benefits and challenges.

To measure effect size, we follow standard statistical practice for within-subjects experimental designs. For task completion time and NASA-TLX scores, we compute pairwise differences between treatment and control conditions for each participant. We then calculate Cohen's d by computing the mean difference between conditions and standard deviation.

Replication Package. All experiment materials, anonymized data, and analysis scripts are available in our replication package [8].

Table 2: Time investment of control (without annotations) and treatment (with annotations) conditions; annotation times only measured in the latter (N=15).

Condition	Mean	Median	SD	Range
Control (WITHOUT)	17.02 min	14.46 min	12.41 min	5.24–47.59 min
Treatment (WITH)	21.13 min	20.41 min	8.85 min	6.46–39.45 min
Annotation Time (min)	1.18 min	1.17 min	0.51 min	0.39–2.43 min
Annotation Time (%)	6.27%	5.29%	3.26%	2.45–14.95%

4 Experiment Results

We now present the results of our controlled experiment. All 15 participants successfully completed both tasks, though the quality and completeness of implementations varied. However, the created annotations were generally well-formed and correctly linked to the intended features with no wrongly annotated features.

4.1 RQ1: Time and Workload Overhead

To answer RQ1 and assess the effect size of adding embedded feature annotations, we measured both the time developers spent creating annotations and their perceived workload using the NASA Task Load Index (NASA-TLX).

Time Investment. Table 2 summarizes the median and mean (M) time investment required to create embedded feature annotations, as well as the standard deviation (SD) and time range. On average, participants spent $\mathbf{6.27\% \pm 3.26\%}$ **of their total development time** (Median: 5.29%) creating feature annotations and editing the feature model. Participants spent an average of 21.13 minutes on tasks with annotations and 17.02 minutes on tasks without annotations. In absolute terms, this corresponds to approximately $\mathbf{1.18 \pm 0.51}$ **minutes** (Median: 1.17 min) of annotation time per task. A breakdown of time investment per participant is shown in Fig. 2. The annotation time as a percentage of total development time ranged from 2.45% to 14.95%, indicating considerable individual variation in annotation speed.

To assess the effect size of adding annotations on overall development time, we computed pairwise mean differences and the corresponding standard deviation between conditions. Additionally, as a robustness check, we verified that annotation activity did not inflate completion time beyond the annotation time itself. The mean difference between conditions was 4.11 minutes (Treatment: M=21.13 min, Control: M=17.02 min). However, participants spent only 1.18 minutes on average creating annotations, which corresponds to 6.27% of their total development time. The difference between the between-condition mean difference and the measured annotation time suggests that experimental order and task-specific factors influenced the overall time investment, as participants spent more time on the task they completed first regardless of condition (Task 1 M=24.52 min, Task 2 M=13.64 min). Still, the overall effect size was small (Cohen's d = 0.265), suggesting that, while participants spent time annotating, this overhead did not substantially increase their total task completion time.

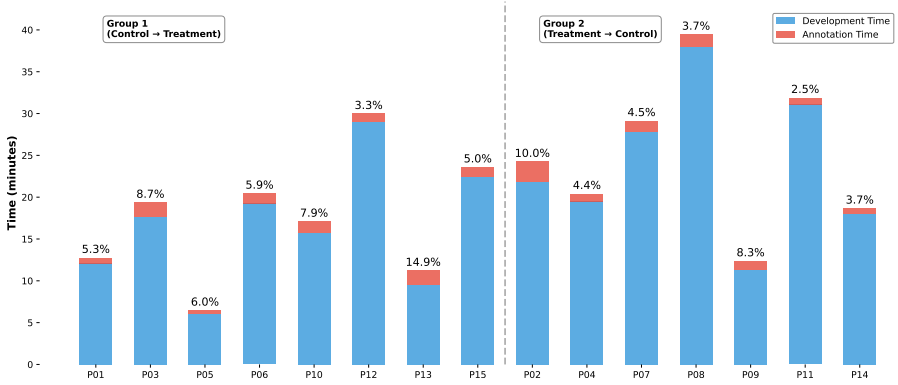

Fig. 2: Time investment per participant showing development time and annotation time as a percentage of total task completion time. Participants are grouped by experimental order: group 1 (Control→Treatment, n=8) and group 2 (Treatment→Control, n=7).

We observed variation between groups: group 1 (who performed the annotation task second) spent $7.13\% \pm 3.58\%$ of development time on annotations, while group 2 (who performed the annotation task first) spent $5.29\% \pm 2.78\%$. However, this difference could reflect task-specific annotation requirements, practice effects, fatigue effects, or individual variation. The order of conditions affected the time investment, with group 1 (Mean time difference: -6.34 min) spending more time during the treatment condition than group 2 (Mean time difference: 16.06 min), suggesting a potential learning effect or task difficulty difference between the first and second task.

Perceived Workload. To assess the effect size of creating annotations on perceived workload beyond time investment, we measured participants' perceived workload using the NASA Task Load Index (NASA-TLX) [23]. The NASA-TLX evaluates six dimensions on a 0–100 scale: mental demand, physical demand, temporal demand, performance, effort, and frustration.

Table 3 presents the raw NASA-TLX scores with no weights applied. We first computed pairwise mean differences and standard deviations in NASA-TLX scores (Control: M=28.9, SD=19.8; Treatment: M=28.0, SD=17.9), and found that effect size was negligible (Cohen's d = 0.06). Figure 3 visualizes the NASA-TLX workload comparison between conditions. Examining individual dimensions, participants reported marginally *lower* mental demand (Control: M=32.0; Treatment: M=30.0), effort (Control: M=31.3; Treatment: M=29.3), and physical demand when working with annotations. Frustration was slightly higher with annotations (Control: M=36.7; Treatment: M=38.7), which aligns with participants' reported challenges around feature naming decisions and determining annotation boundaries (discussed in Sec. 4.2). Perceived performance remained identical across conditions (M=80.7).

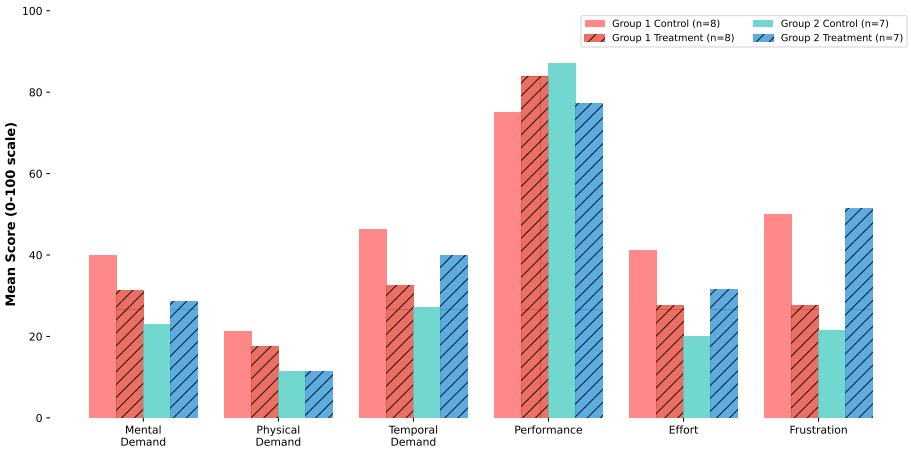

Fig. 3: Barchart of NASA-TLX overall workload scores comparing Control (without annotations) and Treatment (with annotations) conditions (N=15).

Developers invested approximately 6.3% of their development time (mean: 1.18 minutes) in annotation activity, measured directly from screen recordings. A between-condition completion-time comparison yielded a mean difference of 4.11 min (Cohen's d = 0.265), but a task-order effect confounds its interpretation. embedded feature annotations had negligible effect on perceived workload (mean diff = -0.9, Cohen's d = 0.06).

Time and Workload Overhead (RQ1)

4.2 RQ2: Developer Perceptions

To understand how developers perceive the use of embedded feature annotations, we analyzed both quantitative and qualitative survey responses.

Quantitative Perceptions. Table 4 presents developers' responses to Likert-scale questions (1 = Strongly Disagree, 5 = Strongly Agree). The results reveal strong agreement that annotations are valuable for maintenance (M=4.33, 93% agree/strongly agree). Participants also found the annotation process intuitive

Table 3: Mean NASA-TLX dimension scores and overall workload (0–100 scale, N=15). Performance is reverse-coded, so high values indicate greater workload.

NASA-TLX Dimension	Control	Treatment	Difference
Mental Demand	32.0	30.0	-2.0
Physical Demand	16.7	14.7	-2.0
Temporal Demand	37.3	36.0	-1.3
Performance	80.7	80.7	0.0
Effort	31.3	29.3	-2.0
Frustration	36.7	38.7	+2.0
Overall Workload	**28.9**	**28.0**	**-0.9**

Table 4: Developer perceptions of annotation process (Likert scale 1–5, N=15). % Positive indicates agree + strongly agree responses.

Statement	Mean	% Positive
Annotations will be useful for future maintenance	4.33	93%
I found the annotation process intuitive	4.07	80%
I understood the purpose of annotations clearly	4.00	87%
I felt confident in deciding where/how to annotate	3.93	73%
Adding annotations enhanced my understanding	3.53	60%
I would use this in my regular workflow	3.47	53%

(M=4.07, 80% agree/strongly agree) and reported understanding the purpose clearly (M=4.00, 87% agree/strongly agree). Most participants felt confident in deciding where and how to annotate (M=3.93, 73% agree/strongly agree).

However, ratings were more moderate for immediate benefits: only 60% agreed that annotations enhanced their understanding during the current task (M=3.53), and only 53% indicated they would adopt this in their regular workflow (M=3.47). This gap between perceived future value (93%) and willingness to adopt (53%) suggests that developers recognize the long-term benefits but perceive barriers to immediate adoption. When asked directly about their preference, 53% (8/15) preferred working *with* annotations, 13% (2/15) preferred working *without* annotations, and 33% (5/15) reported *no difference*. This preference data aligns with the moderate adoption willingness scores.

Qualitative Perceptions. Participants identified several key benefits of embedded feature annotations through open-ended survey responses.

Faster Code Navigation. Although we did not specifically investigate feature location overhead, the most frequently reported benefit was improved navigation speed when identifying entry points for changes when features were annotated (8/15 participants). P02 noted: *"With annotations I found where to do my code change really fast."* P14 observed: *"Finding certain features within code was easier,"* and P13 appreciated being able to *"browse the different features with usage."* P12 emphasized the efficiency: *"It's easier to just apply annotation than nothing or existing feature flag."* These responses suggest that participants acknowledged the value of annotations as a means to quickly locate features in code.

Code comprehension. Three participants highlighted how annotations helped with understanding the source code. P03 highlighted how annotations enable developers to *"see if actually the code they will add already exists before implementing. Reduces duplicated code."* P10 noted, code with feature annotations is *"easier to follow* than code with *otherwise undocumented features."* This suggests annotations can serve as a mechanism for discovering existing feature implementations before writing new code, and improve overall code comprehension.

Low Overhead. Three participants remarked on the low effort required. P07 commented: *"It doesn't take much effort,"* while P14 noted: *"It doesn't take much work in practice."* P05 mentioned *"the ease of use - selecting the code and then have it wrap with the syntax automatically."*, suggesting that the tool support for creating annotations reduced the overhead of creating feature annotations.

Traceability and Change Impact. Two participants noted the value of traceability and change impact estimation afforded by annotations. P12 stated: *"Traceability is useful when we change existing working code which will help avoid breaking changes."* P03 noted: *"when you change a certain code section, you can easily see where it is used."* These responses suggest that participants recognized the value of annotations for understanding the potential impact of code changes and maintaining traceability between features and their implementations.

Challenges. Participants additionally identified challenges while annotating.

Learning Curve. Three participants mentioned initial confusion that resolved with practice. Although P14 noted the low overhead, they thought it *"probably takes a while to get used to."* P01 explained: *"At the start (task 1) not very clear, but once I understood it a bit better it was rather intuitive."* P15 noted it was *"something new that you need to think about doing."*

Feature Naming and Organization. Five participants mentioned that deciding how to name and organize features within the feature model hierarchy was challenging. P02 claimed challenges were *"only what names to give each annotation."* P09 struggled with *"naming the feature and knowing where to place it in the tree,"* while P07 found it difficult *"finding where in the structure I was and should add to."* These challenges suggest that, while the annotation process itself may be straightforward, the cognitive overhead of determining how to structure and name features within the feature model can be difficult.

Domain Knowledge Dependency. Participants without domain knowledge found annotations less helpful during the initial task. Four participants (P04, P07, P10, P12) had limited domain familiarity. Out of these, P07 observed: *"Every annotation had short, descriptive names, but as someone without domain knowledge, that didn't say that much to me."* This suggests that annotations are most effective when developers share domain understanding.

Tool Integration Issues. Four participants encountered technical issues. P15 reported: *"The code completion didn't work all the time,"* P13 mentioned *"missing keyboard shortcuts that I was used to,"* and P01 noted a *"bug with FM view."*

Overall, participants that preferred working with annotations cited improved navigation, traceability, and reduced duplication as key benefits. They claimed that it *"made it easy to find the code I was looking for"* (P13), that it is *"better to have a feature documentation in some form than none at all."* (P12), and that *"with annotations, I found where to do my code change really fast. Without annotations, I had to search manually for the class."* Participants that preferred working without annotations cited the learning curve, and added complexity to the development process. In particular, they explain that *"it adds on a new layer of complexity marking the code, naming it and putting it correct in the structure."* (P11) and that *"it's something new that you need to think about doing and in my experience that usually leaves a lot of potential for errors/misuse/confusion"* (P15). Finally, participants that reported no difference cited that they were not able to see the benefits of annotations, explaining that *"it's just a comment"* (P09), and *"since no annotation existed before the task there were no differences. If the code had been annotated beforehand, it would have been easier to navigate*

the code to find where to extend or work with the task in hand.", confirming that the benefits of annotations are primarily long-term and may not be immediately apparent during the initial annotation process.

Most developers (93%) viewed annotations as beneficial for maintenance, and 80% found them intuitive. Over half (53%) preferred using annotations for improved navigation, traceability, and reduced duplication, indicated intent to adopt them in regular workflows. However, 7 participants did not, indicating a gap between perceived benefits and actual adoption willingness. Main challenges included learning curve, feature naming, domain knowledge, and tool integration issues.

Developer Perceptions (RQ2)

5 Discussion

We now discuss the implications of our findings on the overhead and how short-term benefits can encourage developers to adopt embedded feature annotations.

5.1 Overhead Investment

Our quantitative analysis revealed that participants spent an average of 6.27% of their total development time on annotation tasks (mean annotation time 1.18 minutes per task). Our statistical analysis showed only a small effect size in development time between tasks with and without annotations (Cohen's d=0.265), suggesting that annotation overhead does not strongly interfere with normal development activities. This finding empirically validates recent cost-benefit analysis [15,27] showing that eager annotation overhead is largely amortized via its benefits, with as few as 3–38% of annotations saving 90–100% of feature-location costs in maintenance scenarios. Our measured annotation time (M=1.18 minutes, 6.27% of development time) confirms that annotation overhead is negligible compared to lazy feature location, which is significantly more expensive.

Importantly, NASA-TLX measurements revealed no significant difference in perceived workload between tasks with and without annotations (Control M=28.9, Treatment M=28.0), demonstrating that using embedded feature annotations does not meaningfully increase developers' perceived workload during active development. The minimal overhead in both time and perceived workload validates embedded feature annotations as a lightweight traceability technique.

While our findings deliver promising evidence of manageable annotation overhead, we did not yet fully explore the whole space of attention investment. Future work should investigate the actual cognitive load of annotation tasks using more fine-grained measures such as eye-tracking, physiological sensors, or think-aloud protocols. Furthermore, a longitudinal study should examine how annotation overhead evolves over time as developers gain experience, how they scale with larger systems and more complex features, and how they interact with different types of development tasks (e.g., bug fixing vs. new feature development).

5.2 Developer Perceptions and Adoption Barriers

The benefits of embedded feature annotations are primarily long-term, focusing on maintenance and evolution. While 93% of participants believed annotations would be useful for maintenance and 80% found them intuitive, the expressed willingness to adopt them was more modest, with only 53% indicating they would incorporate annotations into their regular workflow. This gap between perceived value and adoption intent suggests that long-term benefits alone may be insufficient motivation without immediate, tangible advantages during development, as 60% of participants indicated that annotations do not enhance their understanding of the system.

To encourage adoption, developers need short-term incentives that provide immediate value during their daily workflow. Metrics and visualizations can serve this purpose by helping developers understand features and their interactions in real time [24]. By making features visible during development, such tools aid developers in feature comprehension and decision-making.

Qualitative feedback revealed key insights into adoption barriers. Participants appreciated the plugin's browsing functionality for recovering feature locations and acknowledged how annotations enhance documentation and system comprehension. However, some participants did not fully utilize IDE functionalities (such as "surround with annotation"), resulting in higher annotation times for certain individuals. This suggests that better training, nudging mechanisms, or contextual tutorials could reduce the learning curve and improve efficiency.

File-level annotations posed particular challenges due to the need for precise file and feature naming, which increased navigation overhead. Future improvements could include allowing annotating on the file-level for currently open files, or integrating recommender systems using large language models that learn from previous annotations. Such automation could further minimize overhead while maintaining annotation quality [41].

6 Threats to Validity

Internal Validity. Our within-subjects design with counterbalanced task order controls for individual differences and learning effects. However, the warm-up may not have brought all participants to equal proficiency with embedded feature annotations. Screen recording may underestimate overhead in terms of time by missing cognitive overhead in deciding where and how to annotate.

External Validity. Our 15 professional C# developers from a single organization had substantial experience (M=7.2 years, expertise M=3.73) but low feature modeling familiarity (M=2.4). The 6.27% annotation overhead on our 17,000 LOC industrial system with 39 features may not generalize to larger systems with deeper hierarchies or greater tangling. Our two tasks represent common maintenance activities but not the full spectrum of development work. The controlled experiment sessions differ from real-world contexts with competing pressures and less structured annotation decisions.

Statistical Conclusion Validity. With groups of $n = 7$ and $n = 8$ and high variance in completion times, null-hypothesis significance testing would be severely underpowered, making p-values unreliable guides to the practical magnitude of differences. Our study should therefore be treated as exploratory, with findings motivating rather than confirming effect estimates; a larger study would be required to achieve sufficient statistical power for hypothesis testing.
Construct Validity. Our operationalization of "overhead in terms of time and workload" via task time, annotation time, and NASA-TLX captures multiple dimensions but may miss cognitive overhead of deciding *where* and *how* to annotate, feature naming, and consistency maintenance. NASA-TLX relies on self-report and may not fully capture cognitive load. Additional measures such as Electroencephalography, eye-tracking, or physiological indicators could strengthen validity [2]. We measured the overhead of creating annotations but not downstream benefits, relying on developer perceptions rather than empirical measurement. To explore potential confounds, we computed Spearman rank correlations between annotation time percentage and participant-level factors. Programming expertise ($r = 0.03$) and feature-modeling familiarity ($r = 0.27$) showed only weak associations with annotation time; task order showed the strongest correlation ($r = 0.34$), consistent with the order effect described in Sec. 4.1. We assessed annotation correctness by reviewing participants' screen recordings; all annotations were correctly linked to the intended features, indicating that annotation quality did not explain variation in annotation time.

7 Related Work

Feature Location and Traceability Approaches. Automated feature location techniques using information retrieval, dynamic analysis, and hybrid approaches have been extensively studied [21,50,59,17]. However, these techniques suffer from high false-positive rates and struggle with precision in large industrial systems [1,13,12], requiring substantial manual validation overhead. External traceability databases [49] that link features to artifacts impose high maintenance burdens when code evolves [55,56,58]. Unlike these approaches, embedded feature annotations enable developers to record feature knowledge proactively during implementation while it remains fresh in memory.
Variability Tooling and Annotation Systems. Many tools leverage variability annotations for product-line configuration. Colligens [38] maps C preprocessor directives to feature models, ECCO [22] locates cloned features automatically, VariantSync [46] supports clone-and-own mappings, FeatureIDE [39] loads configurations via preprocessors, and CIDE [29] highlights feature-specific code. These tools generally assume annotations already exist (extracted from preprocessor directives) or focus on optional features in product lines, not addressing the creation overhead when developers annotate features during initial development of new systems or when retroactively annotating existing codebases. Similarly, automation approaches like FeatRacer [41] and Virtual Platform [3,35] aim to reduce annotation burden through machine learning recommendations and annotation-

based cloning, but their cost-benefit remains unclear without empirical baselines for manual annotation overhead.

Empirical Measurement of Traceability Overhead. Prior empirical work on traceability overhead focused primarily on maintenance costs of external databases [55,56] rather than measuring the creation overhead for embedded annotations during active development. Our controlled experiment with professional developers performing real implementation tasks provides the first direct measurement of annotation time and workload, establishing a baseline for evaluating both manual annotation practices and future automation approaches.

8 Conclusion

We investigated the overhead of recording feature traceability during development using embedded feature annotations. While prior work demonstrated long-term benefits through simulation [27,15], and comprehension improvements via visualizations [57], the overhead during development remained unmeasured.

Our controlled experiment with 15 professional developers provides the first empirical evidence that recording embedded feature annotations requires only a small overhead. Annotation time uses only **6.27% of total development time**, and NASA-TLX measurements reveal **no meaningful increase in perceived workload** compared to development without annotations. These concrete measurements validate the key assumption of underlying cost-benefit models: *the upfront investment is small enough to be amortized by long-term benefits.*

Developer perceptions reveal a gap between recognized value and adoption intent. While 93% acknowledged long-term benefits, only 53% expressed willingness to adopt annotations immediately. This suggests that short-term incentives through visualizations and immediate feedback may be crucial for sustained adoption [24]. Our discussion highlights how feature-based visualizations can provide immediate comprehension benefits, creating a positive feedback loop in which developers perceive value despite the added overhead.

Future Work. The advent of AI-assisted code generation tools introduces challenges for feature traceability, as AI-generated code often lacks explicit documentation of its purpose or the features it implements. However, it also presents opportunities to integrate feature annotations directly into the development workflow, potentially reducing the overhead of maintaining traceability. Future work should investigate whether embedded feature annotations can help developers maintain traceability when working with AI-generated code, and whether AI tools themselves can be extended to suggest and generate annotations based on developer prompts or commit messages. We plan to reduce annotation overhead through recommender systems for annotations based on code context, targeting stages where developers invest the most attention. Longitudinal field studies in industry will assess how attention investment evolves with proficiency, how it scales to larger systems, and whether visualizations motivate annotation behavior. Our empirical baseline enables more accurate cost-benefit analyses and provides a benchmark for evaluating automation approaches.

References

1. Abukwaik, H., Burger, A., Andam, B.K., Berger, T.: Semi-automated feature traceability with embedded annotations. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 529–533. IEEE, Madrid, Spain (2018). https://doi.org/10.1109/ICSME.2018.00049
2. Amirova, R., Dlamini, G., Repryntseva, A., Succi, G., Tarasau, H.: Attention and concentration for software developers. IEEE Access **11**, 96196–96205 (2023)
3. Antkiewicz, M., Ji, W., Berger, T., Czarnecki, K., Schmorleiz, T., Lämmel, R., Stănciulescu, u., Wąsowski, A., Schaefer, I.: Flexible product line engineering with a virtual platform. In: Companion Proceedings of the 36th International Conference on Software Engineering. p. 532–535. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2591062.2591126
4. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Berlin Heidelberg, Heidelberg, Germany (2013), https://books.google.de/books?id=4WUnAQAAQBAJ
5. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature Interactions, pp. 213–241 (2013)
6. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)
7. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In: GPCE (2005)
8. Authors: Online appendix: Replication package. https://www.dropbox.com/scl/fo/vzpgdcu9cw36fjtxkf7uy/AAE4JQgHj-s_Fh2AhdZ3QqQ?rlkey=36ooqryselfh9v4bkdk348ecj&st=wkabhhi9&dl=0 (2025)
9. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering **30**(6), 355–371 (2004)
10. Batory, D.: Feature-oriented programming and the ahead tool suite. In: Proceedings. 26th International Conference on Software Engineering. pp. 702–703. IEEE (2004)
11. Behringer, B., Palz, J., Berger, T.: Peopl: Projectional editing of product lines. In: ICSE (2017)
12. ben Othmane, L., Chehrazi, G., Bodden, E., Tsalovski, P., Brucker, A.D.: Time for addressing software security issues: Prediction models and impacting factors. Data Science and Engineering **2**(2), 107–124 (2017). https://doi.org/10.1007/s41019-016-0019-8
13. ben Othmane, L., Chehrazi, G., Bodden, E., Tsalovski, P., Brucker, A.D., Miseldine, P.: Factors impacting the effort required to fix security vulnerabilities. In: Lopez, J., Mitchell, C.J. (eds.) Information Security, Lecture Notes in Computer Science, vol. 9290, pp. 102–119. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23318-5{_}6
14. Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K.: What is a feature? a qualitative study of features in industrial software product lines. In: Proceedings of the 19th International Conference on Software Product Line. p. 16–25. SPLC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2791060.2791108, https://doi.org/10.1145/2791060.2791108
15. Berger, T., Mahmood, W., Abu Zahra, R., Vassilevski, I., Burger, A., Ji, W., Antkiewicz, M., Czarnecki, K.: Cost and benefit of tracing features with embedded annotations. Transactions on Software Engineering and Methodology (TOSEM) (2025)

16. Berger, T., Nair, D., Rublack, R., Atlee, J.M., Czarnecki, K., Wasowski, A.: Three cases of feature-based variability modeling in industry. In: MODELS (2014)
17. Biggerstaff, T.J., Mitbander, B.G., Webster, D.E.: Program understanding and the concept assignment problem. Communications of the ACM **37**(5), 72–82 (may 1994). https://doi.org/10.1145/176579.176597
18. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press/Addison-Wesley Publishing Co., USA (2000)
19. Charness, G., Gneezy, U., Kuhn, M.A.: Experimental methods: Between-subject and within-subject design. Journal of Economic Behavior & Organization **81**(1), 1–8 (2012). https://doi.org/10.1016/j.jebo.2011.08.009
20. Cohen, J.: Statistical power analysis for the behavioral sciences. routledge (2013)
21. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process **25**(1), 53–95 (2013). https://doi.org/https://doi.org/10.1002/smr.567, https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.567
22. Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: The ecco tool: Extraction and composition for clone-and-own. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2, pp. 665–668 (2015). https://doi.org/10.1109/ICSE.2015.218
23. Hart, S.G., Staveland, L.E.: Development of nasa-tlx (task load index): Results of empirical and theoretical research. In: Hancock, P.A., Meshkati, N. (eds.) Human Mental Workload, Advances in Psychology, vol. 52, pp. 139–183. North-Holland (1988). https://doi.org/https://doi.org/10.1016/S0166-4115(08)62386-9, https://www.sciencedirect.com/science/article/pii/S0166411508623869
24. Hermann, K., Martinson, J., Berger, T.: Visualizing feature-oriented software evolution. In: SPLC. p. 136–141 (2025). https://doi.org/10.1145/3744915.3748471
25. Hewett, R., Kijsanayothin, P.: On modeling software defect repair time. Empirical Software Engineering **14**, 165–186 (04 2009). https://doi.org/10.1007/s10664-008-9064-x
26. Jedlitschka, A., Ciolkowski, M., Pfahl, D.: Reporting Experiments in Software Engineering, pp. 201–228. Springer, London (01 2008). https://doi.org/10.1007/978-1-84800-044-5_8
27. Ji, W., Berger, T., Antkiewicz, M., Czarnecki, K.: Maintaining feature traceability with embedded annotations. In: Schmidt, D.C. (ed.) Proceedings of the 19th International Conference on Software Product Line. pp. 61–70. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2791060.2791107
28. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990), http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231
29. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. In: Proceedings of the 12th International Software Product Line Conference. pp. 121–130. SPLC '08, IEEE, Limerick, Ireland (2008)
30. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: Deltaj 1.5: Delta-oriented programming for java 1.5. In: PPPJ (2014)
31. Krueger, J., Calikli, G., Berger, T., Leich, T., Saake, G.: Effects of explicit feature traceability on program comprehension. In: 27th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) (2019)

32. Krüger, J., Gu, W., Shen, H., Mukelabai, M., Hebig, R., Berger, T.: Towards a better understanding of software features and their characteristics: A case study of marlin. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems. p. 105–112. VAMOS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3168365.3168371, https://doi.org/10.1145/3168365.3168371
33. Krüger, J., Berger, T., Leich, T.: Features and How to Find Them: A Survey on Manual Feature Location, pp. 153–172. LLC/CRC Press, Boca Raton, FL, USA (01 2019). https://doi.org/10.1201/9780429022067-7
34. Larman, C., Vodde, B.: Scaling lean & agile development: Thinking and organizational tools for large-scale scrum (2008)
35. Mahmood, W., Calikli, G., Struber, D., Lammel, R., Mukelabai, M., Berger, T.: Virtual platform: Effective and seamless variability management for software systems. IEEE Transactions on Software Engineering **50**(1), 1–31 (2024). https://doi.org/10.1109/TSE.2024.3406224
36. Martinson, J., Jansson, H., Mukelabai, M., Berger, T., Bergel, A., Ho-Quang, T.: Hans: Ide-based editing support for embedded feature annotations. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B. SPLC '21 (2021)
37. Martinson, J., Mahmood, W., Gyimah, J., Berger, T.: FM-PRO: A feature modeling process. IEEE Trans. Software Eng. **51**(1), 262–282 (2025)
38. Medeiros, F., Lima, T., Dalton, F., Ribeiro, M., Gheyi, R., Fonseca, B.: Colligens: A tool to support the development of preprocessor-based software product lines in c. In: Proc. Brazilian Conf. Software: Theory and Practice (CBSoft). Brasilia, Brazil (2013)
39. Meinicke, J., Thüm, T., Schröter, R., Krieter, S., Benduhn, F., Saake, G., Leich, T.: Featureide: Taming the preprocessor wilderness. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). pp. 629–632. IEEE Press, Austin, TX, USA (2016)
40. Microsoft Corporation: Microsoft Teams. www.microsoft.com (2025), www.microsoft.com, accessed: 11 December 2025
41. Mukelabai, M., Hermann, K., Berger, T., Steghöfer, J.P.: Featracer: Locating features through assisted traceability. IEEE Transactions on Software Engineering **49**(12), 5398–5414 (2023). https://doi.org/10.1109/TSE.2023.3290613
42. Olsson, H.H., Alahyari, H., Bosch, J.: Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: SEAA (2012)
43. Parnin, C., Rugaber, S.: Resumption strategies for interrupted programming tasks. Software Quality Journal **19**(1), 5–34 (2011). https://doi.org/10.1007/s11219-010-9104-9
44. Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., Guo, J.: Feature-oriented software evolution. In: Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2430502.2430526, https://doi.org/10.1145/2430502.2430526
45. Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., Valente, M.T.: Feature scattering in the large: A longitudinal study of linux kernel device drivers. In: 14th International Conference on Modularity (MODULARITY) (2015)
46. Pfofe, T., Thüm, T., Schulze, S., Fenske, W., Schaefer, I.: Synchronizing software variants with variantsync. In: Proceedings of the 20th International Systems and

Software Product Line Conference. p. 329–332. SPLC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2934466.2962726, https://doi.org/10.1145/2934466.2962726
47. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: ECOOP (1997)
48. Riebisch, M.: Towards a More Precise Definition of Feature Models. In: Modeling Variability for Object-Oriented Product Lines (07 2003)
49. Robillard, M.P., Murphy, G.C.: Representing concerns in source code. ACM Transactions on Software Engineering and Methodology **16**(1), 3 (2007). https://doi.org/10.1145/1189748.1189751
50. Rubin, J., Chechik, M.: A Survey of Feature Location Techniques, pp. 29–58. Springer, Berlin, Heidelberg, Germany (2013). https://doi.org/10.1007/978-3-642-36654-3_2, https://doi.org/10.1007/978-3-642-36654-3_2
51. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: International Conference on Software Product Lines. pp. 77–91. Springer (2010)
52. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: FOSD (2010)
53. Schwarz, T., Mahmood, W., Berger, T.: A common notation and tool support for embedded feature annotations. In: Capilla, R., Collet, P., Gazzillo, P., Krüger, J., Lopez-Herrejon, R.E., Nadi, S., Perrouin, G., Reinhartz-Berger, I., Rubin, J., Schaefer, I. (eds.) Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B. pp. 5–8. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3382026.3431253
54. Seiler, M., Hübner, P., Paech, B.: Comparing traceability through information retrieval, commits, interaction logs, and tags. In: 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST) (2019)
55. Seiler, M., Paech, B.: Using tags to support feature management across issue tracking systems and version control systems. In: REFSQ (2017)
56. Seiler, M., Paech, B.: Documenting and exploiting software feature knowledge through tags. In: SEKE (2019)
57. Sulír, M., Nosál', M., Porubän, J.: Recording concerns in source code using annotations. Computer Languages, Systems & Structures **46**, 44–65 (2016)
58. Vale, T., de Almeida, E.S., Alves, V., Kulesza, U., Niu, N., de Lima, R.: Software product lines traceability: A systematic mapping study. Information and Software Technology **84**, 1–18 (2017)
59. Wang, J., Peng, X., Xing, Z., Zhao, W.: How developers perform feature location tasks: a human-centric and process-oriented exploratory study. Journal of Software: Evolution and Process **25**, 1193–1224 (2013)
60. Wąsowski, A., Berger, T.: Feature Modeling, pp. 437–457. Springer (2023), https://doi.org/10.1007/978-3-031-23669-3_12