



Gonçalves, P. W., Çalıkılı, G. and Bacchelli, A. (2022) Interpersonal Conflicts During Code Review: Developers' Experience and Practices. In: 25th ACM Conference on Computer-Supported Cooperative Work And Social Computing (CSCW 22), 12 - 16 November 2022, art. 98. (doi: [10.1145/3512945](https://doi.org/10.1145/3512945)).

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

© The Authors 2022. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the 25th ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW 22), 12 - 16 November 2022, art. 98.

<http://eprints.gla.ac.uk/273446/>

Deposited on: 14 January 2022

Interpersonal Conflicts During Code Review

Developers' Experience and Practices

PAVLÍNA WURZEL GONÇALVES, University of Zurich, Switzerland

GÜL ÇALIKLI, University of Glasgow, United Kingdom

ALBERTO BACCHELLI, University of Zurich, Switzerland

Code review consists of manual inspection, discussion, and judgment of source code by developers other than the code's author. Due to discussions around competing ideas and group decision-making processes, interpersonal conflicts during code reviews are expected. This study systematically investigates how developers perceive code review conflicts and addresses interpersonal conflicts during code reviews as a theoretical construct. Through the thematic analysis of interviews conducted with 22 developers, we confirm that conflicts during code reviews are commonplace, anticipated and seen as normal by developers. Even though conflicts do happen and carry a negative impact for the review, conflicts—if resolved constructively—can also create value and bring improvement. Moreover, the analysis provided insights on how strongly conflicts during code review and its context (*i.e.*, code, developer, team, organization) are intertwined. Finally, there are aspects specific to code review conflicts that call for the research and application of customized conflict resolution and management techniques, some of which are discussed in this paper.

Data and material: <https://doi.org/10.5281/zenodo.5848794>

CCS Concepts: • **Software and its engineering** → **Collaboration in software development**; *Software development process management*.

Additional Key Words and Phrases: code review, human factors, interpersonal conflicts, conflict management

1 INTRODUCTION

Code review, is a manual inspection of source code by developers other than the code's author and is considered a valuable tool for finding defects and improving software quality [5]. Unlike the highly structured code inspection formalized almost fifty years ago [37], the most widespread form of code review nowadays (also known as modern code review [9]) is a change-based event, requiring short-term reactions from reviewers and ongoing feedback from colleagues [68]. Through discussion, a decision is reached on whether a code change can be merged into the codebase or whether it should be reworked or rejected [30].

Effective code reviewing requires understanding each others' code and comments and making one's own understandable [9]—challenging activities that pose cognitive load on the developer and reviewer(s) [13, 14, 76]. Moreover, modern code review benefits from discussing competing ideas [9], therefore in addition to misinterpretations about code and comments, a misalignment in developers' goals, priorities, ideas, and perspectives may exist. These features of code review make *interpersonal conflicts* likely to happen [45].

Hartwick and Barki [45] define an interpersonal conflict as “a dynamic process that occurs between interdependent parties as they experience negative emotional reactions to perceived disagreements and interference with the attainment of their goals.”

Interpersonal conflicts have been studied within the context of open-source software (OSS) [39, 40, 48] and Information Systems (IS) development [10]. In the latter context, Barki and Hartwick [10] have described each conflict situation according to three components: *negative emotions*, *cognitive disagreement*, and *interference of behavior*.

Authors' addresses: Pavlína Wurzel Gonçalves, p.goncalves@ifi.uzh.ch, University of Zurich, Binzmühlestrasse 14, Zurich, ZH, Switzerland, 8050; Gül Çalikli, HandanGul.Calikli@glasgow.ac.uk, University of Glasgow, Glasgow, Scotland, United Kingdom; Alberto Bacchelli, bacchelli@ifi.uzh.ch, University of Zurich, Binzmühlestrasse 14, Zurich, ZH, Switzerland, 8050.

An example of a potential interpersonal conflict during code review can be found in the Linux Kernel Mailing List (LKML) [71]. Figure 1 shows an email where the sender comments on a pull request in LKML [79]. This email message contains evidence of:

- (1) *negative emotions* (e.g., “...what makes me *upset* is that the crap is for completely bogus reasons”),
- (2) *disagreements* (e.g., “...the code could easily have been done with just a single and understandable conditional”), and
- (3) *interference* (i.e., writing an email to a mailing list with a harsh language likely to let the pull request’s author to experience negative emotions).

However, the use of harsh language is not the only form of how conflict can take place. For instance, empirical evidence shows that developers can get frustrated if reviewers block their code change requests by withholding the code change’s acceptance too long or by asking for excessive modifications [36]. In general, software practitioners repeatedly refer to the adverse effects of conflicts during code review [32, 65, 74]. For example, OSS developers can stop contributing to a project because of conflicts during the review process [48]. Yet, the literature supports that conflicts can be constructive besides being destructive by bringing change and solutions to problematic situations [31].

Analyzing the aforementioned conflict components (i.e., *negative emotions*, *interference*, and *disagreement*) can help detect when a conflict is happening and is a first step towards measuring the risk of conflicts. Measurement of the conflict components can lead to a proper assessment of the individuals’ perceived level of interpersonal conflicts [10], which facilitates both the assessment of the severity of conflicts and the formulation of appropriate resolution and management strategies. The assessment of the level at which conflicts are perceived is also crucial to evaluate the effectiveness of the conflict resolution and management strategies when put into practice. Moreover, knowing the *conflict focus* (e.g., whether the conflict is about the work to be done or how the work needs to be done) is essential to formulate conflict resolution and management strategies specific to the situation. Hartwick and Barki’s construct of interpersonal conflicts consists of the two dimensions (*conflict components* and *conflict focus*) and provides a basis for in-depth analysis of interpersonal conflicts [45]. Moreover, the development of conflict resolution and management techniques requires understanding the interaction of the conflict process with the conflict’s antecedents and outcomes, which is depicted in the conceptual framework (Figure 2) proposed by Barki and Hartwick [10].

Despite the prevalence of code review in practice and its nature that can spark conflicts, few studies touch upon this topic and only focus on a *single aspect of conflicts* such as pushback [36] and profanity [70, 72]. Conflict resolution with constructive suggestions was empirically identified as an effective strategy to prevent developers from leaving OSS projects [48]. However, studies are needed to further understand how conflicts during code reviews happen and developers’ motivations and behavior in these situations.

In this paper, we present an in-depth investigation of interpersonal conflicts in code review as a theoretical construct and we systematically describe: (1) components and focus of conflicts, (2) antecedents of conflicts, (3) consequences of conflicts, and (4) developers’ strategies for conflict resolution and management. To conduct our investigation, we carried out 22 interviews and analyzed our findings referring to the conceptual framework we adapted from Hartwick and Barki [45].

Our findings indicate that **conflicts are common** during code reviews and **perceived as natural**, mainly due to the exchange of competing ideas. Moreover, using conflicts as a learning and improvement opportunity aids the fulfillment of code review goals; therefore, conflicts are **not to**

Re: [GIT] Networking

From: [REDACTED]
Date: Wed Oct 28 2015 - 05:40:26 EST

- **Next message:** [REDACTED] "[Re: \[PATCH v12 2/6\] fpga: add bindings document for simple fpga bus](#)"
- **Previous message:** [REDACTED] "[Re: \[PATCH\] MAINTAINERS: Start using the 'reviewer' \(R\) tag](#)"
- **In reply to:** [REDACTED] "[\[GIT\] Networking](#)"
- **Next in thread:** [REDACTED] "[Re: \[GIT\] Networking](#)"
- **Messages sorted by:** [\[date\]](#) | [\[thread\]](#) | [\[subject\]](#) | [\[author\]](#)

On Wed, Oct 28, 2015 at 3:32 PM, [REDACTED] <[REDACTED]@xxxxxxxxxxxxxx> wrote:
 >
 > *This may look a bit scary this late in the release cycle, but as is typically*
 > *the case it's predominantly small driver fixes all over the place.*

Christ people. This is just sh*t.

The conflict I get is due to stupid new gcc header file crap. But what makes me upset is that the crap is for completely bogus reasons.

This is the old code in net/ipv6/ipv6_output.c:

```
mtu -= hlen + sizeof(struct frag_hdr);
```

and this is the new "improved" code that uses fancy stuff that wants magical built-in compiler support and has silly wrapper functions for when it doesn't exist:

```
if (overflow_usub(mtu, hlen + sizeof(struct frag_hdr), &mtu) ||
    mtu <= 7)
    goto fail_toobig;
```

and anybody who thinks that the above is

- (a) legible
- (b) efficient (even with the magical compiler support)
- (c) particularly safe

is just incompetent and out to lunch.

The above code is sh*t, and it generates shit code. It looks bad, and there's no reason for it.

The code could *easily* have been done with just a single and understandable conditional, and the compiler would actually have generated better code, and the code would look better and more understandable. Why is this not

Fig. 1. Comment made in the Linux mailing list about a pull request for Linux version 4.3 [79].

be completely prevented but managed. While developing strategies for constructive resolution of conflicts, one should consider that **code review is strongly intertwined with its context**, which can comprise *code*, *developer*, *team*, and *organization*. Moreover, **conflicts in code review require strategies that are specific to its specific context**. Thus, new tools and techniques need to be devised rather than using existing off-the-shelf techniques.

2 BACKGROUND AND RELATED WORK

This section provides general background on interpersonal conflicts in organizations, Information Systems (IS), software engineering, and conflict management. Finally, we provide information on code review practices and the type of code review we consider in this study and we mention studies

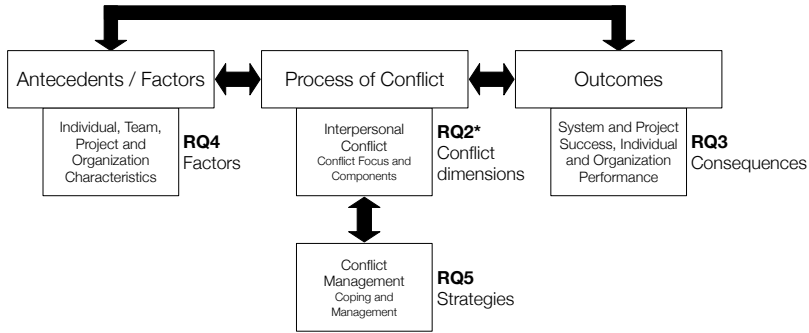


Fig. 2. A Conceptual Framework to Investigate Code Review Conflicts (adapted from Barki and Hartwick Barki and Hartwick [10]).

* The conceptual framework employs the two dimensional construct by Hartwick and Barki [45] to define *Interpersonal Conflict*.

on social aspects of code review related to conflicts and studies addressing specific aspects of code review conflicts (e.g., pushback [36], profanity [70, 72]).

Interpersonal conflict has been studied in diverse fields, including psychology, management, and communication. Despite being an unpleasant experience often with adverse effects, conflicts also present opportunities for improvement and innovation [25].

Studies on apparent conflicts are prevalent in the literature [53]. However, conflicts can also be *hidden*, especially in organizations, due to the need to keep a professional appearance. Dubinkas [35] defines *hidden conflicts* as conflicts that often are invisible to the conflict's own members, which in turn hides such conflicts from the analysis. Analyzing hidden conflicts is crucial to understand the origins and consequences of organizational conflict. Hidden conflicts can threaten or preserve the order in organizations as well as leading to incremental changes in the organizations through the adoption of new routines. Moreover, hidden conflict resolution requires strategies that might be informal and different from traditional ways of conflict intervention [11, 52].

2.1 Conflicts in Information Systems and Software Engineering

There is evidence on the existence of conflicts between IS users and developers [10], and within distributed development teams [6], including OSS development teams [39, 40] and collaborative environments [78], such as Wikipedia [49, 51, 73, 83, 84]. Conflicts can impair collaborators' productivity and the quality of their work [7, 8] and even lead them to leave the project [48]. Moreover, the presence of conflicts is related to lower process satisfaction, decreased system quality, and worse adherence to budget, schedule, and requirements [10], as well as to a lower perception of project's performance and reduced developers' identification with their OSS team [39, 40].

Filippova and Cho [39, 40] categorized conflicts during OSS development as the task, process, norm, and affective conflicts. Differently from Filippova and Cho [39, 40], Hartwick and Barki [45] defined an interpersonal conflict as a two-dimensional construct consisting of *conflict focus* and *components*. They also conducted a survey with IS staff and users to analyze correlations among the *conflict components* (i.e., *negative emotions*, *interference*, and *disagreement*). Their findings serve as the empirical verification of the hypothesis stating that interpersonal conflict comprises all three components. Barki and Hartwick [10] also conceptualized a framework that we adapt to conduct an in-depth analysis of interpersonal conflicts within the context of code review (see Figure 2). The conceptual framework depicts the interaction between the *antecedents* and *outcomes* of conflict as

well as the *process of conflict*. The *process of conflict* consists of *interpersonal conflict*, defined by its two dimensions as well as by *conflict management*.

2.2 Conflict Management

An extensive body of knowledge covers principles and procedures for constructive conflict resolution from the perspective of management [25, 31] and individual coping strategies [16]. Some conflict resolution and management strategies proposed in the management literature are: distributive bargaining, integrative negotiation, joint training, complex-systems change, and de-escalation [16, 25, 31].

Several strategies to conflict handling and management have been investigated in the context of software development, such as compromising, dominating, avoiding, problem-solving, obliging, integrating, smoothing, rational explanation, social encouragement, forcing, confronting, or accommodating [10, 16, 34, 43, 48]. These studies underline that problem-focused, collaborative, and pro-active solutions relate to better conflict outcomes. While such general approaches and strategies have a broad applicability, specific contexts (e.g., OSS development and collaborative work) may require specific strategies for conflict resolution [48].

Formulating the collaborative problem-solving strategies for conflict management in Modern Code Review can have implications for the software engineering practice. Currently, studies in software development derive the conflict management strategies they investigate from the existing literature [10, 34, 43] or manual inspection of software development artifacts, such as pull-request comments [48]. Strategies that Huang et al. [48] derived from pull-request comments can capture only communicational strategies that are directly observable in the review discussion text. However, such strategies cannot capture developers' interpretations and intents or strategies developers might use outside of the review discussions, such as in-person meetings. To address these shortcomings, in this study, we conduct interviews with developers to investigate the strategies they use to prevent and manage conflicts during code review. This approach allows us to capture the subjective motivations and developers' approaches and we expect it to let us discover a broader range of conflict management strategies.

2.3 Modern Code Review

The most widespread contemporary practice of code review (also known as Modern Code Review (MCR) [9, 24]) consists in an informal and lightweight process that focuses on inspecting new proposed code changes rather than the whole codebase [66]. During MCR, a developer (author) submits a code change that other developers (reviewers) inspect. The main goal of this process is to find as many issues as possible in the submitted code change and provide feedback the author needs to address before the change is accepted and put into production [12]. The reviewers and the code change's author engage in an asynchronous discussion: The reviewers can ask for clarifications or recommend improvements to the author, who can reply to the comments and propose improvements. This mechanism can contain the upload of new versions of the code change (i.e., revised patches or iterations), which leads to an iterative process that finishes when all the reviewers are satisfied with the change or decide not to include it in production.

Besides being informal, lightweight, and asynchronous, MCR is tool-based. Among widely used code review tools are Gerrit [3], Microsoft CodeFlow [44], Facebook's Phabricator [4], and Atlassian Crucible [2]. The core component of such code review tools is that the tool facilitates and reports the discussion between the reviewers and the author on the submitted code change [62]. GitHub pull-requests [1] is another medium for MCR, especially popular with OSS projects. In the pull-based software development model, developers who contribute to the project (contributors) can clone the repository to make changes independently. When a set of changes is ready to be submitted to

the main repository, a member of the project's core team (integrator) inspects the changes. If the changes are not satisfactory, the integrator requests more changes the contributor is supposed to implement. Once the integrator assesses the changes to be satisfactory, the contributors' changes are pulled to the main branch. Some OSS projects use mailing lists for code reviews. For instance, Linux Kernel Development heavily relies on mailing lists. A developer should send the code change (patch) to the appropriate subsystem maintainer through at least one of the Linux Kernel-related mailing lists [26, 27]. The patch then usually gets comments from reviewers on how it can be improved. The developer addresses the problems the reviewer points out and informs the reviewer of the planned changes.

Besides code review tools, pull requests, and mailing lists, face-to-face discussions act as supplementary communication channels during code reviews. For instance, MacLeod et al. [57] report that the author and reviewer also engage in face-to-face discussions during code review at Microsoft, while most communication between author and reviewer occurs through the code review tool. Face-to-face discussions can occur in software development environments where teams work on an internal project in a company (e.g., Microsoft developers), including teams working on an OSS project (e.g., Mozilla developers). On the other hand, the code review process relies only on tools or mailing lists in an organically formed OSS community (e.g., contributors to the Linux Kernel project).

Due to the variety in the code review process in different settings and the explorative nature of this study, we do not limit what type of policies developers follow or what communication and tools they use to perform code reviews.

2.4 Social Code Review and Conflicts

Code review comprises social interactions about a technical problem. A number of studies in the software engineering literature focus on non-technical factors involved in the code review process. Bosu et al. [19] showed that the quality of the code under review affects how the reviewers perceive the reliability, expertise, and trustworthiness of the contributor. Moreover, Bosu et al. found that the code review process is a critical practice for creating successful projects' social underpinning because it helps developers form impressions about their teammates, which impact future collaborations besides future code reviews' outcomes. Baysal et al. [15] found that human factors (e.g., developers' experience and reviewer's load) significantly impact code review outcome. Kononenko et al. [54] showed that the number of people involved in a code review is associated with the quality of the review itself. Developers' experience, which was found to be associated with the code review outcome [15, 54], has been also used in code reviewer recommendation techniques [64, 77]. Thongtanunam et al. [77] proposed a code reviewer recommendation technique based on the developers' experience in reviewing files that are stored in close proximity with the code change to be reviewed; while the approach by Rahman et al. [64] is based on developers' experience in certain specialized technologies and external libraries. Bosu and Carver [18] addressed developers' reputation in OSS environments, which is related to developers' experience. Authors found that OSS developers' reputation helps them get faster feedback and make their contributions more likely to be accepted.

Currently, few studies address some aspects of interpersonal conflicts during code review. Egelman et al. [36] focus on pushback, defined as "the perception of unnecessary interpersonal conflict in code review while a reviewer is blocking a change request." Other studies focus on specific aspects related to conflicts such as unfair treatment [42], and profanity and insults [70, 72] during code review. Empirical findings by German et al. [42] indicate that the majority of contributors to OpenStack, a large industrial OSS ecosystem, have experience with unfair treatment during code review. The study by Huang et al. [48] analyses the effectiveness of three conflict management

strategies on 170 GitHub projects' pull-based code review process, namely "rational explanation", "constructive suggestion", and "social encouragement". The authors identify "constructive suggestions" as an effective conflict management strategy after analyzing the review comments and outcome of a survey conducted with the GitHub projects' developers.

3 METHODOLOGY

This section presents our research questions and provides information on the employed analysis, ethics, and data storing besides the information on the interviews and characteristics of the 22 interviewed developers. Additional information and details on the methodology and other appendices are available in the supplementary material [82].

3.1 Research Questions

Both the study by Egelman et al. [36] and the experience of practitioners as reported in publicly available resources [32, 65, 74] suggest that code review is a potential venue for interpersonal conflicts, especially because it requires frequent discussion of competing ideas. Therefore, we start by asking:

RQ1: How do developers perceive and experience conflicts during code review? RQ1 aims to capture how developers perceive conflicts related to code review (*e.g.*, whether conflicts during code review are an issue or whether they are specific in the context of code review).

To analyze conflicts during code review and formulate the remaining research questions, we rely on a conceptual framework of interpersonal conflicts (see Figure 2) adapted from Barki and Hartwick [10]. Human, organizational and technical factors act as antecedents of conflicts and are affected by the conflict outcomes, creating the ground on which further conflicts are (not) happening.

RQ2: What are conflicts during code review like? To answer RQ2, we refer to Hartwick and Barki's two-dimensional construct of interpersonal conflict [45]. For this purpose, we analyze the data to investigate the focus of conflicts and the conflict components.

RQ3: What are the positive and negative consequences of conflicts during code review? Conflicts can also have the potential to be an opportunity for improvement [31]. Therefore, we are interested in whether developers see and experience any negative but also positive consequences of interpersonal conflicts in code review.

RQ4: What factors intervene in the conflict dynamics? RQ4 focuses on understanding which characteristics of code review and its context contribute to conflicts, in terms of emergence, severity, and constructiveness. The conceptual framework by Hartwick and Barki [45] formulates these characteristics as conflicts' *antecedents*. However, the data we gathered from the interviews suggests that these characteristics can play many intervening roles and can also be moderators, mediators, and phenomena co-occurring with the conflicts that are related to how the conflict unfolds. Therefore, we investigate what *factors* make conflicts more likely to happen or be beneficial rather than destructive.

RQ5: What strategies do developers use to prevent and manage conflicts? Problem-focused and collaborative solutions are a way towards positive conflict outcomes [10, 16]. We aim to provide a developer-driven perspective on such problem-solving strategies, specifically in the context of code review. RQ5 investigates the strategies developers adopt and put into practice to prevent and manage conflicts.

3.2 Interviews

To answer our RQs, we conducted an exploratory study by analyzing data we obtained from semi-structured interviews with developers. Appendix A presents the interview structure. Due to the complex nature of conflicts, to gather data for answering our research questions, we estimated each interview to last at least 45 minutes. We continued the interviews until we reached *saturation*.

As this study explores a complex process, we aimed for *code saturation* rather than *meaning saturation* [46]. In other words, we aimed to identify issues and topics related to conflicts to describe the landscape rather than to exhaustively describe and understand all possible insights within the codes and themes. For the code saturation, around the 20th interview, the topics and issues became repetitive (even if details of the narratives could be extended), therefore we interrupted the interviews after 22 complete ones.¹

Furthermore, the saturation is closely related to the sampling methods. Therefore, we have defined a 2-step sampling strategy (Section 3.4) to firstly identify what characteristics of developers provide a broader variety in topics that appear, secondly to hire these diverse profiles [59]. As the interviews were progressing, it became noticeable that the narratives differed significantly, especially in three types of profiles—the different length of experience, different maturity of the code review process, and different environments of conducting code reviews. Therefore, in the second stage of sampling, we have tried to maximize the variety of profiles, covering more or less experienced developers, varying environments (company, Open Source Software community, academic), and more or less standardized modern code review process. We have used these characteristics to compose and describe the study sample (Section 3.4). Finally, once the interviews were done, during the data analysis (described in Section 3.5), we did not identify any undiscovered areas that seemed worthy investigating further.

3.3 Ethics and Data Handling

Before conducting the interviews, each participant signed a consent form (available in the on-line material). We anonymized the interview transcripts before storing them on our institution's server. All non-anonymized data (e.g., interview recordings) was deleted, and the external provider who transcribed the interviews signed a Non-Disclosure Agreement. We also ensured no direct connection between the anonymized transcripts and the consent forms the participants signed. Furthermore, our institution's Human Subjects Committee approved the methodology of our study.

3.4 Sample

We used a multi-stage purposeful sampling method [59]. In the first stage, we used convenience sampling, gathering data from developers who were available from our social and professional network to participate in the study. In the second stage, we used mixed purposeful sampling consisting of snowball and maximum variation sampling to ensure at the same time maximum possible variance in developer profiles with respect to 'Code Review (CR) experience', 'software development environment', and 'CR process maturity'. The sample is described in Table 1 with respect to these characteristics and frequency of experiencing conflicts. Table 2 describes the meaning of the categorical values in Table 1 and reports the number of developers in each category (i.e., 'count'). For example, in terms of 'CR experience' we have three levels: *entry* (novice developers or students), *mid-career* (developers with responsibilities in the software development process), and *supervisory* (for interviewees on a management level or who supervise and mentor others).

¹This number is aligned with the estimates for grounded theory studies, which state that between 20 to 30 interviews should be expected to conduct a saturated study [28].

Table 1. Descriptives of Interviewed Developers

ID	CR Experience	Software Development Environment	CR Process Maturity	Conflict Frequency
D1	Mid-Career	Internal	High	Regular
D2	Entry Level	Internal	Ad hoc	Occasional
D3	Entry Level	Internal	High	Low
D4	Mid-Career	Community, Academic (Researcher)	Limited	Occasional
D5	Mid-Career	Internal	Ad hoc	Regular
D6	Mid-Career	Internal	High	Occasional
D7	Mid-Career	Internal	High	Occasional
D8	Supervisory	Internal, Community	High	Regular
D9	Supervisory	Internal	High	Regular
D10	Mid-Career	Internal	Limited	Occasional
D11	Mid-Career	Internal, Community	High	Low
D12	Entry Level	Academic (Student)	Limited	Regular
D13	Mid-Career	Community, Academic (Researcher)	Limited	Occasional
D14	Mid-Career	Internal, Academic (Researcher)	High	Regular
D15	Supervisory	Internal	High	Regular
D16	Mid-Career	Internal, Community	High	Occasional
D17	Mid-Career	Internal, Community	High	Regular
D18	Mid-Career	Internal	High	Regular
D19	Supervisory	Internal, Academic (Researcher)	High	Regular
D20	Mid-Career	Community, Academic (Researcher, Student)	Limited	Occasional
D21	Supervisory	Internal, Community	High	Regular
D22	Mid-Career	Internal, Community	Limited	Occasional

Table 2. Definitions and count of categorical values describing the developers in Table 1

Characteristic	Category	Count	Description
Experience Level	Entry Level	3	Developers working on reviews within academic projects or in the beginning of their career.
	Mid-Career	14	Developers who have several years of experience, taking responsibility for parts of the code-base.
	Supervisory	5	Developers who apart of codebase take responsibility for people, leadership and management functions and provide mentoring during reviews.
Environment	Internal	18	Mostly stable teams working on an internal project in a company (including teams working on an OSS code) with rather regular and face-to-face contact.
	Community	10	Organically formed open-source community.
	Academic (Researcher)	5	Working in the academia as a researcher or teaching using code reviews;
	Academic (Student)	2	Doing code reviews for educational purposes or small individual projects while studying.
CR Process Maturity	High	14	Code review is an established and regulated part of the software development cycle.
	Limited	6	Code review is a defined process but is not performed as an established part of a full development cycle (e.g., educating students, performing reviews in a small scale company with limited reviewing options).
	Ad-hoc	2	Reviewing code happens on an irregular basis, and the code review process is not well established.
Conflict Frequency	Low	2	Not encountering conflicts or referring to not having much experience.
	Occasional	9	Encountering conflicts 'rarely' or 'sometimes'.
	Regular	11	Generalized experience with conflicts or their repeated and regular presence in code review.

Concerning ‘CR process maturity’ (Table 2), most interviewees (N=14) take part in processes with *high* maturity (i.e., systematic, structured, and integrated into the software development life cycle). Moreover, participants experience conflicts during code reviews at *low* (N=2), *occasional* (N=9), and *regular* (N=11) frequencies.

3.5 Analysis

To explore the conflicts during code review in a broad perspective based on developers' experience, we used Thematic Analysis [23] with a bottom-up approach. This approach is an inductive one where the researcher derives codes and themes from the data content [23]. Unlike the top-down (deductive) method, where the researcher creates codes in advance (based on some concepts, topics, or ideas), in the bottom-up approach, the codes and themes that the researcher derives closely match the data content (Appendix B provides further details).

Before and during the qualitative analysis, we referred to the literature about interpersonal conflicts, code review, and related areas to interpret the data in relation to the existing body of knowledge. To analyze the data, we used a qualitative data analysis software named NVivo 12.²

Thematic analysis consists of six steps [23]: (1) Familiarizing with the data, (2) generating initial codes, (3) searching for themes, (4) reviewing potential themes, (5) defining and naming themes, and (6) writing a report.

Step 1 (*familiarizing with the data*) was used to read through all the interviews and get to know the data. The goal was to create an initial mind map of the main issues and relationships in the data, clear out how to see the conceptual dimensions of interpersonal conflict as defined in the literature [45], and specify what type of data relates to each research question. Throughout the reading, the data was separated into three datasets in order to analyze the RQs individually. Dataset 1 included parts of the interviews that related to RQ1 and RQ2. Dataset 2 included references for RQ2 and RQ4, and Dataset 3 included references to RQ3 and RQ5. The reason was that specifying differences in factors, focus, and disagreement of conflicts was only developing. The overlaps of datasets were resolved during later steps of the analysis.

Steps 2–6 were executed iteratively and separately for each dataset with the help of related researcher memos. During the *initial coding phase* (step 2), dataset overlaps were resolved. Individual pieces of information related to the research questions were formulated and coded in more abstract terms/codes. The initial structure of data started to emerge as well due to multiple research questions and differing levels of abstraction that participants used to express their experience. In step 3 (*searching for themes*), initial codes were grouped into themes on a higher level of abstraction. This phase was important for (dis)confirming emerging themes through finding (dis)similarities between codes. Important questions included: 'What makes these narratives same/different?' 'What would be different if this aspect of the narrative would change?' 'Are these topics related/different/interacting?'

Step 4 (*reviewing potential themes*) consisted of controlling for overlaps and duplication and going back to the raw data in cases when a decision or clarification was needed. We searched for opportunities for merging themes together to simplify the final data structure. During this step in the last dataset, we conducted a control for overlaps of RQs and themes, simplified and unified the presentation of the results to avoid unnecessary complexity and content duplication.

Step 5 (*defining and naming themes*) was done to provide and write final themes' definitions and names while analyzing the entire data and all available context.

While *writing the report* (the paper) in step 6, we focused on presenting not only the resulting themes but also the context in which they are mentioned, and we aimed to convey the story present in the data while respecting both available knowledge and developers' narratives.

3.6 Determining Validity

We used three ways of validation of the analysis that reflect the context of the data collection. The interviews were collected only by one researcher who also did the analysis. Due to the size of the

²<https://qsrinternational.com/nvivo>

data, it was not possible to involve further people who would be deeply familiar with their content. The validation procedures are disconfirming evidence, peer debriefing, and an audit trail-like process [29], further described in Appendix B.

To *(dis)confirm evidence*, attention was paid during the analysis to consider differences between narratives of different participants and situations. We explicitly searched for similarities that would confirm the importance of a theme or disconfirm the current interpretation of the data.

Throughout the analysis, the first author who conducted and analyzed the interviews was discussing the progress and process of the analysis and challenges with data interpretation with a colleague in the research group, who was acquainted with the content of the interviews, performing so-called *peer debriefing*.

After the analysis, materials about the analysis (available in the supplementary material), methodology, and results were collected for conducting an *audit-like process* by a researcher who was not involved directly in the analysis but who was involved in the study design and execution—the third author of the study. The goal was to examine both the process and product of the analysis and determine the trustworthiness of the findings. The researchers discovered no invalid findings through this process.

3.7 Limitations

Despite having followed the described research method rigorously, our study has the following limitations:

- (1) The study is exploratory and focused on uncovering a broad context of conflicts during code review. Even though this process is sufficient to uncover a wide variety of factors, it cannot provide details of the process in which they intervene in the conflict dynamics.
- (2) We adopt a data-driven perspective and collect the experience of developers with interpersonal conflicts during code review. We work with literature from software engineering and other fields throughout the analysis to maximize the utility of knowledge generated from developers' data. However, interpersonal conflicts (also in the organizational environment) are a knowledge-rich area, and there is a substantial body of knowledge that our study cannot fully incorporate. It is our hope that many theory-driven perspectives on this issue might offer other valuable insights.
- (3) We aimed for code saturation, rather than meaning saturation. Aiming for meaning saturation would, in our case, lead to a study that would require a much greater dataset, with many more interviews, and extensive analysis. For example, while we could identify a rich set of factors, we could not describe all its ways of intervention in conflicts.
- (4) We reach a diverse sample of interviewed developers. However, two developer profiles seem to be harder to recruit for such a study: stubborn and rigid developers and developers who support conflicts. The sample contains two developers who see conflicts positively, and one developer defines himself as a 'perfectionist,' but our results are likely not able to cover these two missing profiles.
- (5) The interviews are collected and analyzed only by the first author. To minimize subjectivity and ensure the analysis's validity, the first author performed peer debriefing with the second author, who had an overview of all the sample interviews. Moreover, the third author performed control on the audit trail.

4 RESULTS

Following the aforementioned method, we conducted 22 interviews, manually analyzed 167 pages of transcripts containing 145,496 words, and organized 5,655 individual codes into the themes

presented in the rest of this section. When quoting developers, we use a [DX] notation, where X is the developer's unique identifier reported in Table 1.

4.1 RQ1: How do developers perceive and experience conflicts during code review?

Developers report a sense of normality around conflicts during code review, and all but one interviewee have experienced conflicts during code review at least once.

Developers perceive conflicts as a **normal** aspect of the code review process (D1, D10, D11, D13, D15, D17, D19, D21, D22). Code review conflicts are also **specific**. While some developers do not distinguish conflicts in code review context from other conflicts in the workplace, they also report that conflicts are *easier to happen* during a code review (D9, D18) or it is the *only place for conflicts* (D18) at work. Furthermore, code review conflicts *directly affect developers' work* (D4, D6, D17, D20) (e.g., developers' place in a team, the progress of their work). Code review comprises feedback on, evaluation, and judgment of the developers' work assessed by colleagues. Developers report that code review conflicts tend to get *personal* (D12, D16) and happen in a *specialized context* - discussing code-related decisions with fellow developers (D2, D5, D6, D9, D11, D18, D22). Discussions of one's work and abilities, which might be personally sensitive, occur through asynchronous communication (e.g., code review tools, emails, task managers) that does not primarily allow for personal connection. Code review conflicts are dependent on the relations among individuals who participate in the code review, but they are focused on the code. To summarize, code review conflicts are *non-technical conflicts about technical issues*. As one developer put it: "Usually, the problems are not technical. That's what I learned while working. ... If somebody else is doing [the] research and that thing is really technical and difficult, usually the problem is interacting with that person even if it's about the code" [D15].

Developers also report that conflicts are hard to escape and **hard to anticipate or handle** (D8, D9, D18), and when conflicts happen, developers are unsure of what to do. An interviewee explained: "I don't have a lot of conflicts during code reviews. And when I do it's usually ... I usually don't know what to do" [D8]

In addition, developers find code review conflicts **unpleasant but potentially beneficial** (D6, D8, D12, D14, D15, D17, D22). Conflicts are related to negative emotions. However, conflicts can be constructive and sometimes necessary to find a resolution and meet the code review goals. For example: "You can have a code review with someone senior that doesn't want conflict, and he doesn't do a proper code review. There's no tension at all, but also there's no improvement, so the goal of the code review is not met" [D15].

4.2 RQ2: What are conflicts during code review like?

To answer RQ2, we decompose and analyze code review conflicts referring to the two-dimensional construct by Hartwick and Barki [45]. This construct comprises *conflict focus* and *conflict components* (i.e., *disagreements, interference, emotions*). Table 3 summarizes the themes and sub-themes that emerged for conflicts' focus and components.

4.2.1 Focus of the Conflict. This point addresses what the conflicts during code review are about. Our results can be summarized under the following areas:

Functional aspects. Conflicts on functional aspects comprise what features and how they should be implemented and the completeness of the features' functionality.

Non-functional aspects. Conflicts on non-functional aspects can be about solutions to optimize code performance, security, and maintainability. Conflicts on non-functional aspects can also be about how the code change is implemented and how the code change fits the overall software design. Our findings also indicate that a significant amount of conflicts on non-functional aspects

Table 3. RQ2: Conflicts Focus and Components in Code Review

Dimension	Participant ID	Dimension	Participant ID
CONFLICT FOCUS		CONFLICT COMPONENTS	
<i>Functional Aspects</i>		Interference	
Features	D8	<i>Public</i>	
Incomplete functionality	D2, D12	Open argument	D2, D5, D13, D14, D17, D2
<i>Non-Functional Aspects</i>		Ignoring each other	D1, D6, D11, D12, D19, D21
Design/Big issues	D11, D13, D15, D22	Forcing actions	D5, D7, D11, D12, D21
Security	D11, D12	Blocking actions	D5, D6, D8
Performance	D1, D6, D11, D19	<i>Hidden</i>	
Implementation	D6, D8, D9, D15	Lack of progress	D1, D5-D9, D11, D14
Maintainability and Readability	D1, D2, D6, D7, D11-D15, D18, D19, D21	Proactivity	D1, D6-D9, D13-D19, D21
<i>Bad procedures and practices</i>		Feeling the emotions	D6, D7, D22
Conventions adherence	D4, D6, D7, D10, D12, D15, D21	Emotions	
Unclear requirements	D12	<i>Basic</i>	
Lack of care about the review process	D6, D22	Fear	D4
CONFLICT COMPONENTS		Sadness	D2, D15
Disagreement		Anger	D2, D7, D8, D12-D14, D18, D19
<i>Task definition</i>		<i>Higher</i>	
Definition of done	D5, D6, D8, D9, D11, D12, D19, D21, D22	Anxiety	D7
Goal	D2, D6, D7, D9, D12, D19	Frustration	D5-D7, D11, D18, D20, D21
Priorities	D2, D4, D6, D8-D10, D12, D13, D16, D18-D21	Hate	D12
<i>Quality assessment</i>		Shame	D7
Code quality	D1, D2, D5-D9, D11, D12, D16, D19, D21	Boredom	D21
Review quality	D5-D7, D9, D12, D18, D21	Disappointment	D1, D6, D22
Assessment perspective	D1, D2, D5-D9, D11, D12, D16, D19, D21	Unhappiness	D6
<i>Information availability</i>		Worry	D4, D17
Work constraints	D4, D6, D13-D15, D22	Discomfort	D4, D6, D7, D11, D14
Knowledge and information usage	D1, D5-D7, D11, D12, D16-D19, D21	Awkwardness	D11
<i>Social perception</i>			
Acceptable behavior	D1, D17, D18, D22		
Comparison	D1, D11, D14, D15, D17		
Interpretation	D1, D6, D7, D9, D15, D17, D18		

regard code readability (*i.e.*, code style, adherence to the code style conventions, consistency of the code style throughout the code base).

One interviewed developer explained: “Some people ... take their personal taste on code like if it was of the whole company. If they use two spaces instead of four, it’s like you will see like 50 comments on the code review like «delete to a space, delete to a space, delete to a space». For them, it’s wrong for you it’s not. You cannot make a discussion on it because it doesn’t have positive or negative points. It’s just two spaces or four spaces. It’s nothing. It’s code style. And they take it like super personal, and they impose on you their personal style. This was a complicated thing also to deal with because it has actually a very easy fix which was to impose general code style ... but some people actually didn’t like it as well” [D18].

Bad procedures. Conflicts on inadequate procedures emerge about following standards, guidelines, and conventions on the overall software development process, including code implementation, code reviews, and documentation. One developer explained: “What pisses many people off is: there are guidelines on how to open a code review and how to write things, how to write the problems, so before opening issues or before opening pull-requests, etc. have a look at those guidelines...” [D4].

4.2.2 Disagreement. Disagreement is the cognitive component of conflicts referring to the divergence of conflict participants’ values, needs, interests, opinions, goals, or objectives. We identified disagreements in the following areas:

Task definition. Reported disagreements on task definition are due to misalignments in understanding the *goal* of the review/change, what is (not) part of the task, the *definition of done*, or who is responsible for the task. Developers have different thresholds when a change in the code

is needed or where is the balance between optimal and sufficient code: “Sometimes it’s just like if I am not following the principles of a programming language or something, but it’s not relevant because the code is working and it’s secure, and it’s functional, but if I am not following the rules of the coding language it doesn’t matter. It’s not relevant for the project, and I feel like it’s mean. It is not relevant for the code review” [D12].

One main factor related to disagreements on task definition is developers’ *prioritization* of what to include in the code implementation or feedback. For instance, a developer might prioritize fixing an issue that might seem irrelevant to the other.

Quality assessment. Differences in the perception of the *code quality* can spin a conflict; also, the perception of what is *useful and relevant* in a code review might differ. As one developer put it: “When I was writing the code, I already included in it all the things she was mentioning, and it seemed to me totally irrelevant. To the extent that I felt like the comments of the people are there for nothing” [D6].

It can be problematic when developers view the code from a different *perspective* (e.g., as code author or reviewer). There can be misalignment in assessing the code complexity to estimate the effort and resources needed for the code’s implementation or review. Problems can also occur due to differences in the abstraction levels. For instance, a developer might base the discussions on code-level without an insight into the overall software design or vice versa.

Information availability. Our findings also indicate disagreements based on differences in the information available to each conflict party. Developers use different resources to obtain *knowledge*, which might cause issues during code review when those information resources disagree. Experience-based knowledge might be incompatible with the knowledge developers gain by studying current practices and state-of-the-art and can also be outdated. Besides, misunderstandings can emerge when knowledge about the software system and architecture or procedures agreed upon within the project team or organization are not available to all developers.

Developers also might lack mutual understanding of their colleagues’ *work constraints*. These limit the developer’s ability to react to requested changes. Developers report struggling with having sufficient time, energy, or other resources to deal with potential problems. A reviewer can make requests that are infeasible or not desirable for the developer simply because the reviewer is either unaware or inconsiderate of the developer’s work constraints.

One developer explained: “Imagine I work on a feature. I am late. I’ve been working for three or four days, my boss is putting pressure on me to deliver and here comes [name of person] and gives me feedback that will require two days of work. You are not going to accept it very well” [D15].

Social perception. Social perception refers to inferring others’ intentions, thoughts, or personality traits based on our perception of their observable behaviors (e.g., verbal or non-verbal communication) [33]. However, the process is prone to errors and misinterpretations [41]. One of the (*mis*)*interpretations* developers mentioned in the interviews is whether one views the feedback as assessing the quality of the code or the developer who implemented the code. The perception of the latter can make the debate more personal. Furthermore, developers can interpret the intention to help identify their insufficiency to solve the problem at hand. “And the other person (not necessarily) wants to fix it or give feedback so that it is better, but I perceived it in the moment like it is not good enough. And that can not hurt you but it tells you: «Well, you could have done it in the first attempt.»” [D6].

Some developers can *compare* themselves to others. Developers also might compete in who is right or who is the better programmer. Furthermore, each individual has a different perception of *acceptable behavior*, which might lead to conflicts. Like so, developers step over what others consider an unacceptable interference with their actions and intentions: “Then they say, «I did

not like this, I changed it, I sent it.» Which I find strange. That’s too much when somebody touches your work. It’s really not nice” [D1].

4.2.3 Interference. Interference is the behavioral component of conflicts, which comprises conflict parties’ actions and how the conflict can manifest. We report different interference types during code review based on the conflicts’ categorization as **public** and **hidden**, referring to the literature [11, 35, 52, 53] we previously mentioned (Section 2). Public conflicts include behaviors directly interfering with the attainment of one’s goals, whereas hidden conflicts are not expressed directly.

Public conflicts. We identified four types of behavior during a code review that can interfere with the attainment of the developer’s goals: *heated argument*, *blocking*, *forcing*, and *ignoring*. Developers can *block* other developers from achieving their goals. For instance, blocking can be in the form of pushback on change acceptance, as also reported by Egelman et al. [36], or simply not allowing a developer to make changes that the developer sees as necessary: “I want to remove some ... blocks of code and they don’t let me do it” [D5].

Forcing developers to do something they do not find correct is also another type of such interference: “When they force me to do something wrong it’s very rare now, but sometimes I do it, I do it as fast as I can so I can jump again for the right way” [D5].

Our findings also indicate *ignoring* and disregarding the opinions and requests of the counterparty as other interference.

Hidden conflicts. In this category, we present a list of symptoms that are characteristic of conflicts. Furthermore, these symptoms of conflicts can also be present in public conflicts. Developers report that, in some situations, there are no obvious signs of a conflict in the communication or behaviors, but the participants *can detect the emotions*. One developer explained: “... when I notified her about it few times, and it was still happening to her, that she did not pay attention, I could see she is not happy with that. But I cannot say she said: «No, I will not do it because I don’t want.» ... In terms of the comment, you cannot really tell” [D6].

Hidden conflicts during code review are accompanied by *proactivity* towards finding a solution. Developers use various strategies like reminding others what should be done or sneaking into code reviews they do not need to be part of to have more control over the codebase. Continuing the discussion outside the code review tool using face-to-face communication, involving other colleagues or a manager, or addressing the issue during a team meeting can all be a symptom of a conflict. Another indication of proactivity during code review is writing long and explanatory comments. Conflicts can also be accompanied by a ‘ping pong’ of comments resulting in a long discussion *lacking progress*: “We talk about it for an hour, and nothing is discussed, nothing will happen, and everyone is still in their stance” [D1].

4.2.4 Emotions. Emotional response to a stimulus (e.g., a discrepancy between needs and goals) is a natural process. Table 3 lists the negative emotions developers reported during the interviews. Emotions can be categorized as basic and higher [22]. **Basic emotions** (e.g., fear, anger, sadness) can be activated by unconditioned stimuli and lead to automatic, involuntary reactions. A learning process determines **higher emotions**. For instance, people learn what situations make them frustrated or ashamed. Therefore, experiencing higher emotions can also be altered by the learning process [22].

Emotions that arise during code review include higher emotions alongside basic emotions since code review requires higher cognitive processes such as reasoning, analysis, and decision-making and is a socially complex situation. As a result, although negative emotions during code review cannot be avoided entirely, developers can learn new ways to interpret the situations resolving higher negative emotions. For instance, developers can learn to accept feedback, give constructive

Table 4. RQ3: Positive and Negative Consequences of Conflicts

	Negative	Participant ID	Positive	Participant ID
Code	Lower code quality	D5, D8-D11, D21	Better code quality	D2, D4-D6, D8
	Worse maintainability	D11, D21	Better maintainability	D11
	Following bad practices	D5	Standardization	D1, D2, D5, D7, D18
Developer	Loosing trust in the review process	D1, D4, D6, D7	Initiating changes	D2
	Disengagement	D1, D4, D6-D8, D12, D15	Generating new ideas and solutions	D17, D19
	Lower productivity	D1, D5, D11, D17, D21	Getting own changes approved	D5
	Leaving/Loosing job	D4, D11, D17, D21, D22	Professional growth	D1, D2, D4, D5., D10, D11, D13, D16
	Reviewer/Reviewee Selectivity	D1, D8, D9, D15, D18, D19	Improved relationship towards work	D5, D8, D14
	Negative relationship towards work	D1, D2, D5-D7, D9, D10, D12, D15	Positive emotions	D2, D5, D17
	Negative emotions	D1, D2, D6, D7, D12, D17, D18		
Team	Impaired collaboration	D1, D2, D5, D6, D8, D9, D11-D18	Better collaboration	D11, D14-D17
	More conflicts	D8, D18, D21	Better communication	D4-D7, D14-D16, D18, D21
Organization	Bad reviews	D17	Better rentability	D5
	Delays	D4, D6, D9, D10, D21	Better working conditions	D5
	Financial losses from underutilized workforce and bad practices	D5, D12		

feedback, and learn about their peers' communication styles. Moreover, understanding negative emotions can provide information about what needs to change to mitigate them. For instance, frustration appears when a person encounters obstacles in a striving path towards their goals [17]. On the other hand, anxiety can yield as one might not know what to do in a specific situation or feels insecure about his abilities to meet particular demands [50].

4.3 RQ3: What are the positive and negative consequences of conflicts during code review?

Our findings indicate that conflicts impact code reviews, the **code**, the **developer** being part of the review, the **team** developer interacts with, and the **organization** the developer works for. These four are nested layers that are related to each other. To illustrate how a conflict's consequence on one layer can impact other layers, we firstly present the following scenario for conflicts' negative consequences based on themes and relationships (also in Table 4) we identified in the analysis:

Code. A team experiences conflicts on poor coding practices. Due to the struggle to improve them, the team produces code with lower quality and maintainability issues. Moreover, developers keep following poor coding practices. One developer explained: "They just bypass me go to the team leader for example and okay «it's working, yeah, go» and this contaminates the code basically" [D21].

Developer. Due to conflicts, developers report to lose trust in the code review process and disengage from contributing to code review discussions or the project. Disrupted code review discussions lead to poor decisions followed by increased developers' stress and negative emotions. Developers' motivation and productivity deteriorates due to the conflict and loss of trust in the fairness and efficiency of the code review process (e.g., no trust in the feedback received or in colleagues). As one developer put it: "I think that your views of the other members of the team

change. That you get an opinion like, «That guy’s a bit weird» or «He thinks too much about himself,» or «He thinks that everything he does is good,» and then there is this grumpiness in the team. When it comes to solving something, you do not want to deal with that person because you already feel he will criticize you strangely. That you do not trust the review anymore, as it should be” [D1].

Team. The conflict leads to further issues with collaboration in the team. Developers start to have negative impressions about their colleagues, their relationships get colder, and the team creates a hostile environment which breeds further conflicts: “I think that impacts a bit in other conflicts because if I have a conflict with someone in the code, this conflict ... I will bring in the other conflict too, so I think it impacts on the other conflict and more conflicts. At some point you just go out and get crazy and start fighting.” [D21].

Organization. As a result, the organization might suffer financial losses from an underutilized workforce, following lousy coding practices, higher turnover of employees and contributors, delays in deployment, or losing clients.

As mentioned in Section 4.1, developers report that conflicts during code reviews can be unpleasant yet beneficial. Developers also report the importance of resolving conflicts and turning them into an opportunity to learn or improve. Therefore, to illustrate how a conflict’s consequence in one layer (e.g., code) can impact other layers (e.g., developer, team, organization), we also present the following scenario for conflicts’ positive consequences based on themes and relationships we identified through data analysis:

Code. Developers resolve the conflicts that arise due to poor coding practices by the collaborative formulation of new coding conventions in the team. Consequently, code quality and maintainability improve. Through the confrontation of ideas, new creative solutions are found.

Developer. Developers learn from each other and their own mistakes, leading to improved technical skills (e.g., coding) and knowledge in addition to improved communication and decision-making skills. One interviewee explained: “I can realize that I am doing something wrong. I can learn from it, and it will be better for everybody. It will be better for me, that I will make it better, and it will be better for the application and it will be better for the other developers who will come and work on it because it will be better planned, better fixed and so on” [D11].

Team. Successful handling of the conflict results in more fruitful discussions and better collaboration within the team. Developers improve how they communicate with each other, their mutual understanding, learn about others’ opinions, and accept criticism: “From personal experience [with] code reviews I think if you have tension and you solve the issues you are in a better place than when you started because people understand you better and they are more comfortable with you. If you let things escalate and you don’t talk, and then you want that person out of work. It’s very complicated” [D15].

Organization. The organization profits from better use of their investments in their workforce and better working conditions that arise from the discussing practices followed in the organization.

4.4 RQ4: What factors intervene in the conflict dynamics?

Factors are the characteristics of the environment that play a role in the presence and severity of conflicts—they are predictors, moderators, or possible areas of prevention for conflicts. Table 5 presents a summary of the factors developers mentioned during the interviews. Similar to explaining conflicts’ consequences (Section 4.3), we explain factors under the following categories:

Code and review. High code complexity and low code quality make it challenging for developers to solve issues efficiently, building tension and frustration. Reviews challenging due to *code quality*, *documentation quality*, *review size*, including documentation and patch size and *system*

Table 5. RQ4: Factors Involved in Conflicts during Code Review

Code and Review	Participant ID	Developer	Participant ID
<i>Change</i>			
Challenging review	D1, D6, D7, D9, D11, D12, D14, D18-D20	Awareness of behaviour in the conflict Engagement	D6, D14, D15, D17, D19 D1, D2, D5-D15, D17-D21
Change importance	D1, D2, D7, D11, D13, D15-D17, D19	Personality	D1, D2, D4-D15, D17-D19, D21, D22
Quality of the change code and solution	D6, D9, D12, D22	Profiting from the review	D1, D2, D5, D9, D13-D16, D20-D22
Review size	D1, D11, D15, D16, D18-D20	Developer role	D6
Type of code	D22	Review role	D1, D8, D11, D14, D20, D22
<i>Codebase</i>			
Code quality	D1, D5, D6, D9, D11-D13, D15, D16, D19, D21	Psychological state	D1, D2, D5, D7, D15, D18, D22
Documentation quality	D6, D16	Workload	D1, D2, D6-D9, D11-D18, D21, D22
System complexity	D7, D19, D20	<i>Experience</i>	
<i>Review policy</i>			
Flexibility	D1, D6-D10, D18	Hierarchy	D3, D6, D7, D9-D11, D14-D19, D21, D22
Frequency of reviews	D6, D7, D10	Learning curve	D1, D4, D6-D10, D12-D15, D17, D1, D20-D22
Systematic reviewing	D2	Priorities	D6, D10, D11
Value of review work	D7, D16, D22	Reviewer experience	D1, D4, D6-D10, D14, D15, D20
<i>Review process</i>			
Binary output	D7, D14, D16	Self-confidence	D1, D4-D7, D9, D10, D12, D14, D17-D19, D21
Availability and stability of reviewers	D1, D2, D6, D7, D10, D11, D16, D17	<i>Team</i>	
Solutions on par	D6, D7, D16, D19	Organization	
Lack of information on how to decide	D1, D6, D7, D11, D13, D16, D18	Culture	D1, D3-D7, D9, D10, D15, D21
Subjectivity of decisions	D11, D13-D16, D18, D19, D21	Development process maturity	D1, D2, D4, D7, D9, D11, D13, D14, D16, D17, D19-D21
<i>Team</i>			
Dependency on others	D5-D10, D17, D18, D20	Leadership quality	D5, D6, D9, D11
Distance and Diversity	D1, D2, D4-D6, D8-D11, D13, D14, D16-D21	Status	D2, D4, D6, D10, D11, D13, D17
Majority	D1, D5, D8, D14, D15, D18, D21	Size	D6, D8, D9, D11, D21
Size	D7, D18	<i>Communication</i>	
Team composition	D9, D11, D14, D18, D20, D21	Environment	
<i>Communication</i>			
Constructive feedback	D1-D9, D11-D19, D21	Internal	D8, D11, D13, D14, D16, D21, D22
Mode	D1, D4, D6, D7, D10, D17, D18, D22	Community	D8, D11, D13, D15, D16, D21
Misunderstanding	D1, D2, D4, D6, D8, D12, D15, D17, D18, D22	Academic	D12, D13, D19, D20
<i>Cooperation</i>			
Agreement on ways of communicating	D14, D15, D17, D18	<i>Cooperation</i>	
Common interest	D11, D17, D18, D20	Team	
Relationship quality	D1-D4, D6-D9, D11, D13-D19, D22	Organization	

complexity also create a challenge to understand and communicate the issues in a code change and to propose optimal solutions. One developer said: “This could be a problem if you have super challenging code reviews and you know you can do it, but it costs you a lot of time. I mean this can be a stress factor as well especially linking it to workload” [D7].

Depending on the *patch priority* and *severity of the issue*, developers determine whether it is worth entering a disagreement: “Well, if more people are discussing it, one can leave without a problem. But if he is one of those people who wrote or reviewed it, he could attend the discussion. And if it’s something serious, like a bug, I would not do anything like that” [D1].

Flexibility to delegate the review and choose reviewers determines whether developers get stuck in a potentially unpleasant situation. Code reviews occurring with high *frequency* and intensity of contact among developers can lead to tension. Moreover, *lack of value* put on code reviews either by individuals or by a team and company can result in poor software development (e.g., coding, code review) practices and consequential tension and conflicts. One interviewee recounted: “one developer ... was known for not caring about the code reviews. That he always writes it somehow and then sometimes even does not do pull-request on GitHub and just sends the commit or he makes the PR, but he merges it by himself. So there was a discussion about either we have some rules, or we don’t. And somebody always repeated this: «Let’s do it properly» and (*name*) just did it in his way again” [D10].

The *binary output* of a review (i.e., ‘Accept’ or ‘Reject’) can be too simplistic and harder to accept. Furthermore, the *availability* and *stability* (i.e., low turnover) of reviewers who could help clear up the situation or bring additional information facilitate decision-making during code reviews.

Developer. Different *developer roles* are related to different **code and review** characteristics thus making conflicts more or less likely to happen. For instance, full-stack developers might engage in code reviews more frequently and systematically than testers. Developers also have their own motivations during code review. *Being aware* of differences in developers’ roles and motivations or one’s own mistakes is a step towards the resolution and management of conflicts during code reviews. Developers’ priorities might differ depending on their *role* as the author or reviewer of the code change (e.g., to get things done vs. to optimize the code). The authors might also feel an attachment to their code and a knowledge of the rationale and process behind the code’s implementation, which reviewers might not always have. The *engagement* in one’s work (e.g., emotional attachment to the code as author) is crucial for active participation in the discussions but also may make developers more sensitive to negative feedback.

Developers who experience *autonomy*, have a feeling of accomplishment, and room for freedom are more engaged to spend *effort in doing a good job*, follow good practices, implement high-quality code, listen to others, and have productive discussions during code reviews. To deal with a conflict, it is also helpful to *see profits* of the conflict (e.g., learning something new, seeing the working result): “Of course everyone can have conflicts and if the guy explained to me that I was doing something wrong and he explained to me in the way that I understand it ... it’s a positive conflict. Otherwise, it’s useless.” [D21]

Psychological state (e.g., mood, fatigue, stress) can lower developers’ ability to deal with negative feedback or the need to rework some changes. *Workload* and deadlines contribute to the work-related stress and consequential behavior in the review discussion.

Developers’ *personality traits* such as openness to change, resilience, perfectionism, sociability, emotionality, individuality, and conscientiousness affect how they engage in the review, discussion, and conflicts: “If someone is stubborn, what can a man do?” [D1].

Developers’ *experience* introduces hierarchy - more experienced developers having more authority and responsibilities. Developers report that novices need to build up self-confidence and improve coding skills and knowledge. Novice developers also need to gain experience in communicating

their decisions and opinions about code and the code’s implications for the project and handling different types of people, and how to get them on the same page. One developer explained: “I accept that the reviewers have more experience than me. So I take the advice. There can be a conflict in the moment when both of the people are confident that their approach is good and the other person sees it differently.” [D10] However, experienced developers might become rigid, slow to change, and over-confident in their knowledge. The novice developers can offer an up-to-date perspective and knowledge that can get lost due to the authority and unwillingness of experienced developers to adjust or novices’ inability to discuss with experienced developers and confront them. Conflicts arise between experienced developers as well, but rather in the form of expert competition.

Team. *Distance and diversity* in a team (e.g., geographical, cultural, personal, professional, organizational) can bring varied perspectives and create challenges in finding common ground and communicating clearly. Developers mention that team members should have good relationships, agreement on ways of communicating, and shared interest (e.g., shared focus, shared code ownership, team effort towards a common goal). However, shared goals and good relationships do not always lead to constructive conflict resolution. For instance, the following may result in non-constructive conflicts: unproductive team habits, or the *majority* of the team members not following proper software development practices or not being open to change. Moreover, how team members are *dependent* on each other (e.g., how much the outcome and speed of the code review affect their work) and getting stuck with people who are hard to cooperate with can also lead to conflicts. Misunderstandings among team members are also the main challenge during code review [9]. Issues during communication might arise when team members’ ability to explain and understand different perspectives is low. As one developer put it: “It becomes complicated when there is, of course, a misunderstanding because there is one person who thinks about one reality and the other person has another reality, and the two don’t match, and you maybe don’t understand why the other person is making jokes and why the other person is screaming at you.” [D17]

Developers also report the importance of *constructive feedback*, which has the following characteristics: (1) understandable (i.e., explains reasons for decisions and suggestions with supporting examples); (2) actionable (i.e., specific on what to do and how to do it); (3) focused on the topic (i.e., specific on what is wrong prioritizing vital issues); and (4) includes positive points. In addition, to facilitate the communication of ideas, developers mention the importance of using *synchronous* and *in-person* communication where the exchange becomes fast and more elaborate.

Organization. A software development environment creates a setting in which interpersonal interaction happens. Developers in our sample (see Table 1) reported important differences, especially between stable teams *internal* to a company and the OSS *community*. *Internal* teams are paid for their job. They have more strictly set deadlines, higher developers’ dependency on company hierarchy, and their motivations are affected by the need to keep the job and income. Moreover, internal team members mostly work in close contact with each other (e.g., in-person meetings, informal communication over coffee), which provides: (1) more context information; (2) more communicational cues and opportunities for clarification; (3) long-term development of inter-personal relationships. One developer said: “Most of the cases the conflicts for me are with colleagues rather than volunteers. I think it’s also because you have way more interactions with your colleagues rather than with the volunteers, at least in the projects I am involved with. We have many volunteers, but every one of them is only contributing very little” [D8].

Community members contribute to an OSS project voluntarily. In a community, interactions form organically, and connections are loose. Due to the voluntary contribution to the project, there is less pressure on community members who can leave problematic projects.

Organization size with a possible increase in the hierarchical or physical distance among developers also introduces complexity in communication. Besides, the *quality of leadership* in an organization is related to problems during code review. Understanding superiors who support the teams' needs and goals facilitate smooth code reviews. Moreover, an organization's *status*, such as the potential for growth or being understaffed, determines how the organization needs to become appealing to developers and how developers feel motivated or stressed.

Maturity of the software development process facilitates clarity of communication among developers. Moreover, automated processes, proper testing, following standards or conventions, and using tools (e.g., static analyzers) can resolve some issues in advance, preventing the waste of resources during code reviews: "If people discuss things that can be automated, it means their process is not set up efficiently ... It's just my vision of how not to waste too much resources on things you don't really have to waste resources on" [D14].

Organizational culture provides for developers an agreement on how to communicate with and treat each other during code reviews. Besides valuing high-quality code, improvement, and innovation, developers mentioned the following values that organizations can incorporate into their culture: equality, respect, fairness, trust, tolerance, safety, openness, and cooperation rather than competitiveness. One interviewee stated: "it's not forbidden to make mistakes in our team so if someone makes a mistake, he is not afraid that someone else will tell him «hey, you made a mistake you are an idiot,» but more like «hey, I think we could do this in another way»" [D7].

4.5 RQ5: What strategies do developers use to prevent and manage conflicts?

We describe the strategies developers employ to prevent and manage conflicts. We also present a detailed categorization of these conflict prevention and management strategies in Table 6.

Acknowledging the conflict. Developers mention the importance of acknowledging conflicts when they occur. Identifying conflict participants' emotions, needs, and goals helps clarify the situation and initiate discussions to find solutions. However, depending on the emotions' intensity, developers can postpone the discussions or next action steps. One developer said: "The way we have to sort it is to take him for a date in the parlor downstairs and discuss it there in a more calm way like take out some stress, go down, have a beer and discuss it with him" [D18].

Active involvement in conflict resolution. Developers initiate de-escalation procedures, acquire knowledge by asking questions, and formulate solutions to resolve conflicts constructively. Developers also formulate lessons learned by explicitly stating the potential benefits of the situation. "This was our way to cope with negativity. So it was to interact more and ask for more details and ask for more information why is this so bad so that we have a learning moment from the community" [D20].

Adapting to individuals. To prevent and address conflicts, developers try to understand the other developers' needs and feelings and adjust their behavior accordingly. The ability to adjust to individual needs is related to how close the developers are to each other and underpins the importance of good relationships within the team. One interviewee explained: "There is a small number of people that you have to do reviews with. So that you can adapt somehow to their way of communication and you understand when they are getting angry, you understand how they work as people so how to calm them down, how to convince them, you understand which arguments work and which arguments they don't like." [D17]

Automation and standardization. As mentioned in Section 4.2.1, our findings also indicate a significant amount of conflicts on code readability (i.e., code style, adherence to the code style conventions, consistency of the code style throughout the codebase). Moreover, some conflicts emerge about **bad procedures** (e.g., not following standards, guidelines, and conventions on the

Table 6. RQ5: Strategies to Prevent and Resolve Conflicts

Strategies	Participant ID	Strategies	Participant ID
Acknowledging the conflict		Adapting to individuals	
Addressing the conflict	D1, D6, D14, D15, D17, D18	Individual approach	D4, D13, D15, D17, D18, D20
Taking a break	D1, D6, D11, D13, D14, D17	Showing interest	D4, D13, D15
Opening the issue up for a discussion	D1	Trying to understand the feelings of others	D15, D18
Expressing needs and opinions	D8, D13-D15, D17, D18, D22	Adjusting own behaviour to others	D18
Expressing emotions	D4, D14, D17, D18, D22		
Reporting issues to authorities	D9, D17, D18		
Active involvement in conflict resolution		Automation and standardization	
Being active towards resolution	D17	Create and update standards and conventions	D7, D11, D14, D16, D18-D20
De-escalation	D9, D17	Automate what is possible	D7, D11, D13, D14, D16, D18, D19, D21
Formulating solutions	D20	Assign a responsible	D14
Formulating lessons learned	D20-D22	Keep workflows flexible	D14, D16
Preparation		Third-party intervention	
Getting to know the code	D1	HR intervention	D9, D18
Preparing what and how to communicate	D17	Giving a decision	D17, D19
Clarify differences	D6, D7-D9, D15-D17	Managing work effort	D1, D8, D9, D11, D17
Do not develop ideas in isolation	D9	Moderating communication	D8, D17, D18
Plan steps	D1	Setting priorities	D10
Agree on implementation	D7, D9	Team reorganization	D11, D16, D20
Send a change suggestion	D9, D19	Providing explanations	D6
Clear requirements	D1, D2, D7, D8		
Setting a process against stagnation		Training	
Creating space for discussion	D1, D2, D6, D9, D11, D16, D17, D19, D20	in Code of Conduct	D5, D18, D21
Deciding on a team level	D1, D11, D13, D16, D17, D21	in Company standards	D5, D18, D21
Synchronizing	D6, D7, D9-D11, D15-D19, D21	in the Software system	D5
Setting priorities	D9, D11, D12, D16, D17, D19, D20	in Constructive feedback	D9, D17
Gathering more knowledge	D1, D4-D8, D11, D12, D14-D18, D20	in Management	D13
Assigning new tasks	D6		

overall software development process). Developers report that adopting standards and automation can help prevent and resolve conflicts. One developer said: “The team should be composed in such a way that you automatize maximum of possible things. Starting with the styling and going as far as you can. And if this is not done properly, it can spin more and more conflicts.” [D11] Even though automation and standardization facilitate communication, the workflows should not become rigid. Furthermore, developers also reported that it is helpful to have team members assigned to monitor the state and assess the need to update the standards, automation, and procedures used in the team.

Preparation. To align the team’s vision about the code change and prevent potential communication issues, developers prepare themselves for code review by clarifying the requirements and agreeing in advance on the code change implementation. Preparing for what and how to communicate during code reviews can also facilitate the prevention and management of conflicts during code reviews. An interviewee explained: “I see the code review as some sort of the last step in the overall human interaction around developing the project. So many steps need to happen before in order to bring people on the same page.” [D9]

Set a process against stagnation. To avoid or resolve conflicts, developers set a process to resolve situations when the discussion is stagnating, for example, by taking discussion from code review tools into in-person meetings to align needs and expectations and give feedback on how the team

members should cooperate. To avoid lengthy discussions over the review tools, developers also employ pair programming. Another strategy is to make decisions based on set priorities (e.g., code simplicity over complexity, authors' opinion over alternatives on par, solution requiring the least effort). When the discussion stagnates, developers also gather knowledge from external resources, documentation, project history, and other developers to support their claims or get needed expertise. To obtain decision power, developers also involve the whole project team or field experts. Moreover, developers can employ democratic voting or list the pros and cons of proposed solutions to facilitate decision-making.

Third-party intervention. In some cases, third parties other than conflict participants get involved in the conflicts' prevention and resolution. Developers sometimes involve Human Resources (HR) to resolve serious situations. Managers can also get involved and clear the situation by: (1) making the final decision, (2) providing rationale behind the decisions made, (3) setting priorities for the software development process (e.g., code changes to implement/review), (4) moderating communication among conflict's participants, (5) adjusting workload among team members, (6) assigning responsibilities to individual team members, (7) providing instruction on how to proceed, and (8) reorganizing the team.

Training. To get developers on the same page, organizations offer training on the software system, code of conduct, and conventions during onboarding. Developers also mention the usefulness of training on management and giving constructive feedback. "I think the only way to actually sort this out is to explicitly train people to give constructive feedback all the time" [D9].

5 DISCUSSION

In this section, we discuss the main findings of our study and their implications for practitioners, researchers, and educators, as well as tool design. We first summarize our main findings:

Conflicts are common and perceived as natural. From RQ1, we found that 21 developers (*i.e.*, all but one) experienced conflicts during code reviews. Nine stated to perceive conflicts as a natural and expected part of code review, and that code review may be the only environment for interpersonal conflicts at work.

Developers reported several reasons for the conflict-prone nature of code reviews (Section 4.1) particularly that (1) developers address personally sensitive issues (e.g., developers' skills, quality of developers' code) in an impersonal environment (e.g., code review tools, emails, task managers) and (2) discuss technical issues through a socially complex process. These two complementary angles suggest that conflict resolution strategies should focus on code reviews' technical and social nature. Similar difficulties to point (1) have been reported for distributed teams that need to communicate over distance and use tool-supported ways of communication [47].

Even though these findings could be limited to our sample developers, we have no reason to think they are caused by sampling bias (we used unrelated sampling criteria—Section 3.4). Therefore, it seems reasonable to derive that interpersonal conflicts during code reviews are an important topic to handle from both a research and a practical perspective.

Conflicts are not to be prevented but managed. Previous studies [36, 70, 72] mainly focused on the negative aspects of code review conflicts. Our results align with those by Barki and Hartwick [10] in their investigation on conflicts in IS development. In particular, that conflicts are related to a decreased developers' productivity and well-being, lower process satisfaction and system quality, as well as worse adherence to budget, schedule, and requirements. However, our findings for RQ1 and RQ3 indicate that interpersonal conflicts during code review can also be constructive. In particular, our analysis revealed how conflicts could be a learning opportunity and bring novelty and additional value.

Both positive and negative consequences of conflicts are closely related to the fulfillment of code review goals such as finding defects, code improvement, and knowledge transfer [9]. On the one hand, all of the review goals are threatened by the presence of conflicts; on the other hand (as also reported in RQ1), conflicts might be a means to actually achieve them in an environment lacking an exchange of competing ideas.

Developers actively use strategies to prevent and address conflicts when they are happening. These strategies show similarities with constructive conflict resolution strategies proposed in psychology and management literature (Section 2.2), such as active involvement in conflict resolution or setting priorities [25]. However, some strategies, such as automation and standardization, are specific to software development and particularly to code review.

Huang et al. [48] emphasized the effectiveness of constructive suggestions as a conflict management strategy. Developers in our study also mention the importance of constructive feedback as a factor involved in the conflict dynamics. Our interviewees also propose ways of conflict management that lead to active involvement in situation resolution and clearing out priorities and next steps in the review and software development process. These findings, together with the evidence that code review conflicts happen naturally and developers cannot avoid them, indicates that further research can be designed and conducted on constructive conflict resolution specific to the context of code review.

Code review and its context are strongly intertwined. When it comes to interpersonal conflicts, code review and its context (*i.e.*, code, developer, team, organization) are strongly intertwined. Indeed, as reported when answering RQ3 (Section 4.3), social interaction problems during code review can negatively affect code, developers, development teams, and organizations in addition to the code review process. These findings align with Hartwick and Barki [10] by showing that conflicts result in decreased developers' productivity and well-being, lower process satisfaction, system quality, worse adherence to budget, schedule, and requirements.

Furthermore, as a response to RQ4 (Section 4.4), we reported factors related to code, developer, team, and organization that can be fruitful areas of intervention to support social interactions. Our findings describe many technical and non-technical factors identified in literature that play a role in software development, influence source code quality or code change acceptance such as: (1) code and review related factors (*e.g.*, patch size, patch priority, code complexity [15], understandability [69], requirements quality [56], documentation clarity [21]); (2) developer-related factors (*e.g.*, experience, engagement [15]) and (3) team and organisation level factors (*e.g.*, relationship quality [80], leadership [81]).

Conceptually, destructive interpersonal conflicts are a form of community smells, which are communication issues leading to communities' inability to tackle software development problems. Research has shown a relationship between community smells and code smells [60], similarly to the consequences of conflicts for the codebase as well as the team and organization that we found (Section 4.3). Conflicts can also lead to sub-optimal decisions and solutions regarding performance, security, or maintainability resulting in technical debt [67]. Moreover, team and organization-level factors listed in Section 4.4 can contribute to social debt, which is the additional cost that occurs when strained social and organizational interactions get in the way of smooth software development and operation [75]. Therefore, our study corroborates that further research on the role of the reported factors can bring new insights into the relation between community smells and code smells and between social and technical debt.

Conflicts in code review are also domain-specific. Our results for RQ1 indicated that conflicts during code reviews are *non-technical conflicts about technical issues* and that code review generates also conflicts that are specific only to this context. For this reason, even though some conflict management strategies can be borrowed from psychology and management literature (*e.g.*,

acknowledging conflicts and adapting to individuals), as reported in RQ5, additional strategies—specific to code review conflicts—must be investigated and brought into practice (such as automation and standardization). In other words, existing off-the-shelf solutions/approaches to handle conflicts are not enough for the code review context.

5.1 Decomposing Conflicts

To understand and navigate conflicts successfully, it is crucial to decompose them into their individual components. This decomposition enables the understanding of the situation on a more fine-grained level and the identification of strategies to address specific and situational needs for conflict prevention, management and resolution. One of the main goals of our study (answered in RQ2) was to describe conflicts during code review using the two-dimensional construct by Hartwick and Barki [45], which consists of *conflict focus* and *conflict components* (i.e., *disagreements*, *interference*, *emotions*). Unlike disagreement and emotions, interference (i.e., behaviors, symptoms) is observable, directly serving as indicators of more risky situations that can lead to conflicts. However, the behaviors of others are subject to the interpretation of an individual. If the individual has no negative emotions, there are no conflicts [45]. Therefore, it is crucial to look for early signs of emotions for conflict resolution and management besides interference. Understanding emotions can help devise solutions to prevent their occurrence, hence conflicts in the future. To summarize, our findings can help:

- (1) identify areas needing improvement or additional knowledge for the review and communication to be successful through analyzing *focus of conflict*;
- (2) understand the *disagreement* where developers' cognition differs to unify or clarify their perspectives for effective conflict resolution;
- (3) analyze developers' communication during code reviews to detect signs of communication difficulties or inappropriate behaviors (i.e., *interference*) to detect the risk of conflicts happening;
- (4) conduct sentiment analysis [38] on data gathered from code review tools to detect and identify negative *emotions*, which can provide clues on the developers' needs that can be addressed for conflict resolution or prevention.

5.2 Implications

The results presented in this paper can be translated into implications for practitioners and management, as well as for educators and researchers. The results can also help inform the design of tools that better support the communication among developers during code reviews.

5.2.1 Practitioners. Our findings present (1) various forms of conflicts (i.e., communication issues, symptoms, interfering behavior, differences in perspectives), (2) factors related to conflicts, and (3) strategies for conflict prevention and management that can serve practitioners as an inspiration on how to detect and prevent or manage conflicts during code review. We also identified that developers' technical skills (e.g., program comprehension and analysis) and soft skills (e.g., communication, cooperation, engagement in constructive criticism) described in the literature [55] could facilitate the prevention, management, and resolution of conflicts. Therefore, training to improve these skills has the potential to prevent conflicts. Organizations should offer practitioners training during onboarding on the software system, conventions, and standards used in the organization for software development to minimize conflicts. Moreover, organizations should design and conduct training that improves practitioners' soft skills (e.g., communication, cooperation, giving constructive feedback).

5.2.2 Tool Design. Tools can help identify and mitigate conflicts during code review.

- (1) *Provide conflict risk metrics.* This study identified numerous symptoms of conflicts and related factors that can be used to build a metric suite to facilitate the detection of conflict-prone code review environments. Some of the symptoms we identified relate to the code and change (code and documentation quality, review complexity), and some relate to the characteristics of the communication (lack of progress, constructive feedback), team (size, composition), project, or their review process (frequency of reviews). Analyzing these factors can bring insights into how conflict-prone a specific review is. Egelman et al. [36] use such metrics to identify pushback in code reviews. Furthermore, text analysis (e.g., Sentiment Analysis [38]) can be used to estimate the emotions in the review and provide recommendations on which reviews or teams need communication improvement, conflict resolution, or management intervention. Furthermore, the text-based computational analysis offers mechanisms to study conflicts from other angles, using the content of the text and employing different machine learning techniques [58].
- (2) *Detect stagnation points.* Conflicts can be indirectly observed through discussion lacking progress. Tools can act as a facilitator of progress by detecting stagnation in the discussion and nudging developers in using some strategy (see Table 6) to resolve the stagnation point.
- (3) *Support switching to synchronous communication.* Code review conflicts are dependent on mutual understanding and can get personal, but they happen through tools and asynchronous communication. Developers mention the importance of discussing more complicated issues in person and having faster feedback on other developers' understanding. Therefore, tools can acknowledge this shortcoming by directly requesting a meeting or a synchronous peer review.
- (4) *Assist in expressing priorities and issue tracking.* Conflicts in code review arise based on misalignment in goals and priorities related to the change and task. Tools can further support the expression of importance, goals, and priorities of the review and issues detected in it, for example, by assigning priorities to reviews and comments.

5.2.3 *Educators.* Code reviews should be an integral part of software project courses. While designing project courses, educators should also consider the points we summarized for practitioners above (e.g., preparations, simplicity and clarity, cooperation). For instance, conducting code reviews during course projects can serve as an experiential learning technique [63] to facilitate students' communication and collaboration skills besides technical skills (e.g., requirements elicitation, software design, coding). Furthermore, students need support to develop sufficient communication skills to formulate and defend technical decisions and provide feedback to their colleagues in a constructive way. Software project courses should also provide students with the practice of adhering to software development guidelines, standards, and coding conventions.

Developers reported the importance of using social skills (e.g., communication, cooperation) as factors involved in interpersonal conflicts (see Section 4.4). Developers find **training** on giving constructive feedback and other topics useful for conflict prevention, resolution and management. Literature suggests that developers need the ability and willingness to share knowledge, constructively criticize, make and fulfill agreements in the team and learn from other team members [55]. Skills can be trained to improve productivity and collaboration including conflicts [25].

5.2.4 *Researchers.* Conflicts' consequences can severely affect not only the code review process but also a codebase, any involved developers and teams, as well as an entire organization (Section 4.4). Based on these findings, an interesting research path is to investigate interpersonal conflicts during code review through the lens of the *gestalt field theory* in psychology literature [61]. Field theory is the observation that any situation's meaning is determined by the relationship between the thing we are focusing on (e.g., code review) and the context it occurs within (e.g., code, developer,

team, organization). According to this theory, it is not sufficient to fix individual factors to correct a dysfunctional situation. Rather the solution lies in finding a new, functional equilibrium in the field. Therefore, fixing interpersonal conflicts would require finding a balance between the code, review, developer, team, and organization-specific factors that facilitate a constructive discussion in a stable environment.

5.3 Future Work

This study acted as an initial broad description of potential forms of conflicts during code review, their context, and effects. There are several areas that require further research work to understand conflicts and mitigate their potentially negative consequences:

- (1) Quantitatively confirm the conflicts' frequency and severity in code reviews.
- (2) Quantitatively investigate positive and negative consequences of conflicts with a focus on their relation to the fulfillment of the review goals.
- (3) Further describe the mechanisms of involvement of individual factors.
- (4) Investigate the effectiveness of in-person modes of communication and their integration in code review tools.
- (5) Investigate whether developers are sufficiently informed and equipped to handle conflicts during code reviews.
- (6) Use proposed components and factors to build conflicts risk assessment metrics.
- (7) Quantitatively investigate the efficiency and effectiveness of conflict management strategies developers use and their applicability.
- (8) Identify further conflict management strategies in the literature and assess their utility in the context of code review.

6 CONCLUSION

In this study, we presented a qualitative investigation on interpersonal conflicts in code review. By interviewing 22 developers with a range of expertise and from different environments, we found that conflicts are common in code review and perceived as a natural part of the task. However, not all conflicts lead to poor outcomes—some improve various social and technical aspects of software development. Therefore, conflicts do not necessarily need to be prevented but rather managed. Developers describe a number of factors that affect interpersonal conflicts during code review, as well as several problem-focused strategies that can be used to identify and understand conflicts, thus leading to better conflict management. We hope that the insights we have discovered and presented in this paper will form the basis for further research on this prominent aspect of code review and software engineering.

ACKNOWLEDGMENTS

P. Wurzel Gonçalves and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

- [1] 2021. <https://docs.github.com/en/github/collaborating-with-pull-requests>, title = GitHub Docs: Collaborating with Pull Requests.
- [2] 2021. Atlassian Crucible: Collaborative Code Review. <https://www.atlassian.com/software/crucible>.
- [3] 2021. Gerrit Code Review. <https://www.gerritcodereview.com/>.
- [4] 2021. Phabricator. <https://secure.phabricator.com/book/phabricator/article/introduction/>.
- [5] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. 1989. Software Inspections: An Effective Verification Process. *IEEE Softw.* 6, 3 (May 1989), 31–36.

- [6] Mashael Saeed Alqhtani and M Rizwan Jameel Qureshi. 2014. A Proposal to Improve Communication between Distributed Development Teams. *International Journal of Intelligent Systems and Applications (IJISA)* 6, 12 (2014), 34–39.
- [7] Ofer Arazy, Oded Nov, Raymond Patterson, and Lisa Yeo. 2011. Information quality in Wikipedia: The effects of group composition and task conflict. *Journal of Management Information Systems* 27, 4 (2011), 71–98.
- [8] Ofer Arazy, Lisa Yeo, and Oded Nov. 2013. Stay on the Wikipedia task: When task-related disagreements slip into personal and procedural conflicts. *Journal of the American Society for Information Science and Technology* 64, 8 (2013), 1634–1648.
- [9] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.
- [10] Henri Barki and Jon Hartwick. 2001. Interpersonal conflict and its management in information system development. *MIS quarterly* (2001), 195–228.
- [11] Jean M. Bartunek and Robin R. Read. 1992. The Role of Conflict in a Second Order Change Attempt. In *Hidden Conflict in Organisations: Uncovering Behind-the-Scenes Disputes*, Deborah M. Kolb and Jean M. Bartunek (Eds.). Sage Publications, Beverly Hills, Chapter 5, 116–142.
- [12] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. 2016. Factors Influencing Code Review Processes in Industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 85–96.
- [13] Tobias Baum and Kurt Schneider. 2016. On the Need for a New Generation of Code Review Tools. In *Product-Focused Software Process Improvement*, Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen (Eds.). Springer International Publishing, Cham, 301–308.
- [14] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 329–340. <https://doi.org/10.1109/ICSME.2017.28>
- [15] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 122–131.
- [16] Hasida Ben-Zur. 2009. Coping styles and affect. *International Journal of Stress Management* 16, 2 (2009), 87.
- [17] Leonard Berkowitz. 1989. Frustration-aggression hypothesis: examination and reformulation. *Psychological bulletin* 106, 1 (1989), 59.
- [18] Amiangshu Bosu and Jeffrey C Carver. 2014. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–10.
- [19] Amiangshu Bosu, Jeffrey C Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2016), 56–75.
- [20] Marian Carcary. 2009. The research audit trial—enhancing trustworthiness in qualitative inquiry. *Electronic Journal of Business Research Methods* 7, 1 (2009), pp11–24.
- [21] Adelina Ciurumelea, Sebastian Proksch, and Harald C Gall. 2020. Suggesting Comment Completions for Python using Neural Language Models. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 456–467.
- [22] Jason A Clark. 2010. Relations of homology between higher cognitive emotions and basic emotions. *Biology & Philosophy* 25, 1 (2010), 75–94.
- [23] Victoria Clarke, Virginia Braun, and Nikki Hayfield. 2015. Thematic analysis. *Qualitative psychology: A practical guide to research methods* (2015), 222–248.
- [24] Jason Cohen. 2010. Modern Code Review. In *Making Software*, Andy Oram and Greg Wilson (Eds.). O’Reilly., Chapter 16, 329–338.
- [25] Peter T Coleman. 2012. Constructive conflict resolution and sustainable peace. In *Psychological components of sustainable peace*. Springer, 55–84.
- [26] The Kernel Development Community. 2022. *Submitting Patches: The Essential Guide to Getting your Code into the Kernel*. Retrieved Jan 12, 2022 from <https://01.org/linuxgraphics/gfx-docs/drm/process/submitting-patches.html>
- [27] Jonathan Corbet. 2016. *Why Kernel Development Still Uses Email*. Retrieved Jan 12, 2022 from <https://lwn.net/Articles/702177/>
- [28] John W Creswell. 2002. *Educational research: Planning, conducting, and evaluating quantitative*. Prentice Hall Upper Saddle River, NJ.
- [29] John W Creswell and Dana L Miller. 2000. Determining validity in qualitative inquiry. *Theory into practice* 39, 3 (2000), 124–130.

- [30] Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software* (2021), 110951.
- [31] Morton Deutsch. 1994. Constructive conflict resolution: Principles, training, and research. *Journal of social issues* 50, 1 (1994), 13–32.
- [32] Erik Dietrich. 2020. *How to Deal with an Insufferable Code Reviewer*. Retrieved September, 2020 from <https://daedtech.com/insufferable-code-reviewer/>
- [33] Ap Dijksterhuis and John A Bargh. 2001. The perception-behavior expressway: Automatic effects of social perception on social behavior. In *Advances in experimental social psychology*. Vol. 33. Elsevier, 1–40.
- [34] Madeline Ann Domino, Rosann Webb Collins, Alan R Hevner, and Cynthia F Cohen. 2003. Conflict in collaborative software development. In *Proceedings of the 2003 SIGMIS conference on Computer personnel research: Freedom in Philadelphia—leveraging differences and diversity in the IT workforce*. 44–51.
- [35] Frank Dubinskas. 1992. Culture and Conflict: The Cultural Roots of Discord. In *Hidden Conflict in Organisations: Uncovering Behind-the-Scenes Disputes*, Deborah M. Kolb and Jean M. Bartunek (Eds.). Sage Publications, Beverly Hills, Chapter 8, 187–208.
- [36] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Maggie Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Pushbackacterizing and Detecting Negative Interpersonal Interactions in Code Review. (2020).
- [37] M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211. <https://doi.org/10.1147/sj.153.0182>
- [38] Ronen Feldman. 2013. Techniques and applications for sentiment analysis. *Commun. ACM* 56, 4 (2013), 82–89.
- [39] Anna Filippova and Hichang Cho. 2015. Mudslinging and manners: Unpacking conflict in free and open source software. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. 1393–1403.
- [40] Anna Filippova and Hichang Cho. 2016. The effects and antecedents of conflict in free and open source software development. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. 705–716.
- [41] David C Funder. 1987. Errors and mistakes: Evaluating the accuracy of social judgment. *Psychological bulletin* 101, 1 (1987), 75.
- [42] Daniel German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. 2018. "Was My Contribution Fairly Reviewed?" A Framework to Study the Perception of Fairness in Modern Code Reviews. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 523–534.
- [43] David H Gobeli, Harold F Koenig, and Iris Bechinger. 1998. Managing conflict in software development teams: A multilevel analysis. *Journal of Product Innovation Management: AN INTERNATIONAL PUBLICATION OF THE PRODUCT DEVELOPMENT & MANAGEMENT ASSOCIATION* 15, 5 (1998), 423–435.
- [44] Michaela Greiler. 2021. How Code Reviews Work at Microsoft. <https://www.michaelagreiler.com/code-reviews-at-microsoft-how-to-code-review-at-a-large-software-company/>.
- [45] Jon Hartwick and Henri Barki. 2002. Conceptualizing the construct of interpersonal conflict. *Cahier du GReSI no 2*, 04 (2002).
- [46] Monique M Hennink, Bonnie N Kaiser, and Vincent C Marconi. 2017. Code saturation versus meaning saturation: how many interviews are enough? *Qualitative health research* 27, 4 (2017), 591–608.
- [47] Pamela J Hinds and Diane E Bailey. 2003. Out of sight, out of sync: Understanding conflict in distributed teams. *Organization science* 14, 6 (2003), 615–632.
- [48] Wenjian Huang, Tun Lu, Haiyi Zhu, Guo Li, and Ning Gu. 2016. Effectiveness of conflict management strategies in peer review process of online collaboration projects. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. 717–728.
- [49] Bernard Jacquemin, Aurélien Lauf, Céline Poudat, Martine Hurault-Plantet, and Nicolas Auray. 2008. Managing conflicts between users in Wikipedia. *arXiv preprint arXiv:0805.4754* (2008).
- [50] Paul Keedwell and RP Snaith. 1996. What do anxiety scales measure? *Acta Psychiatrica Scandinavica* 93, 3 (1996), 177–180.
- [51] Aniket Kittur, Bongwon Suh, Bryan A Pendleton, and Ed H Chi. 2007. He says, she says: conflict and coordination in Wikipedia. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 453–462.
- [52] Deborah Kolb. 1992. Women's Work: Peacemaking in the Organisations. In *Hidden Conflict in Organisations: Uncovering Behind-the-Scenes Disputes*, Deborah M. Kolb and Jean M. Bartunek (Eds.). Sage Publications, Beverly Hills, Chapter 3, 63–91.
- [53] Deborah Kolb and Jean M Bartunek. 1992. *Hidden conflict in organizations: Uncovering behind-the-scenes disputes*. Vol. 141. Sage.
- [54] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE international conference on software maintenance and evolution*

- (*ICSME*). IEEE, 111–120.
- [55] Barbara Linck, Laura Ohrndorf, Sigrid Schubert, Peer Stechert, Johannes Magenheimer, Wolfgang Nelles, Jonas Neugebauer, and Niclas Schaper. 2013. Competence model for informatics modelling and system comprehension. In *2013 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 85–93.
- [56] Julie Yu-Chih Liu, Hun-Gee Chen, Charlie C Chen, and Tsong Shin Sheu. 2011. Relationships among interpersonal conflict, requirements uncertainty, and software project performance. *International Journal of Project Management* 29, 5 (2011), 547–556.
- [57] Laura MacLeod, Michaela Greiler, Margaret-Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code reviewing in the trenches: Challenges and best practices. *IEEE Software* 35, 4 (2017), 34–42.
- [58] Seraphine F Maerz and Cornelius Puschmann. 2020. Text as data for conflict research: A literature survey. *Computational Conflict Research* (2020), 43–65.
- [59] Anthony J Onwuegbuzie and Nancy L Leech. 2007. A call for qualitative power analyses. *Quality & quantity* 41, 1 (2007), 105–121.
- [60] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering* (2018).
- [61] Malcolm Parlett. 1991. Reflections on field theory. *British Gestalt Journal* 1, 2 (1991), 69–81.
- [62] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW (2018).
- [63] Felicia Patrick. 2011. *Handbook of Research on Improving Learning and Motivation*.
- [64] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 222–231.
- [65] Philipp Ranzhin. 2019. *I ruin developers' lives with my code reviews and I'm sorry*. Retrieved September, 2020 from <https://habr.com/en/post/440736/>
- [66] Peter C. Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (Saint Petersburg, Russia) (ESEC/FSE 2013)*. 202–212.
- [67] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* 102 (2018), 117–145.
- [68] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 181–190.
- [69] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 417–427.
- [70] Daniel Schneider, Scott Spurlock, and Megan Squire. 2016. Differentiating Communication Styles of Leaders on the Linux Kernel Mailing List. In *Proceedings of the 12th International Symposium on Open Collaboration*. 1–10.
- [71] Jasper Spaans. 2022. *Linux Kernel Mailing List Archive*. Retrieved Jan 12, 2022 from <https://lkml.org/>
- [72] Megan Squire and Rebecca Gazda. 2015. FLOSS as a Source for Profanity and Insults: Collecting the Data. In *2015 48th Hawaii International Conference on System Sciences*. IEEE, 5290–5298.
- [73] Róbert Sumi, Taha Yasseri, et al. 2011. Edit wars in Wikipedia. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*. IEEE, 724–727.
- [74] Nate Swanner. 2015. *Linux creator Linus Torvalds had a meltdown over a pull request, and it was awesome*. Retrieved September, 2020 from <https://thenextweb.com/dd/2015/11/02/linux-creator-linus-torvalds-had-a-meltdown-over-a-pull-request-and-it-was-awesome/>
- [75] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans Van Vliet. 2015. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications* 6, 1 (2015), 10.
- [76] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 168–179. <https://doi.org/10.1109/MSR.2015.23>
- [77] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.
- [78] János Török, Gerardo Iñiguez, Taha Yasseri, Maxi San Miguel, Kimmo Kaski, and János Kertész. 2013. Opinions, conflicts, and consensus: modeling social dynamics in a collaborative environment. *Physical review letters* 110, 8 (2013),

088701.

- [79] Linus Torvalds. 2015. *Linux Kernel Mailing List Archive*. Retrieved Jan 12, 2022 from <http://web.archive.org/web/20210223033541/http://lkml.iu.edu/hypermail/linux/kernel/1510.3/02866.html>
- [80] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th international conference on Software engineering*. 356–366.
- [81] Vathsala Wickramasinghe and Sahana Nandula. 2015. Diversity in team composition, relationship conflict and team leader support on globally distributed virtual software development team performance. *Strategic Outsourcing: An International Journal* (2015).
- [82] Pavlina Wurzel Gonçalves, Gül Çalıklı, and Alberto Bacchelli. 2022. *Data and Material for 'Interpersonal Conflicts During Code Review'*. <https://doi.org/10.5281/zenodo.5848794>
- [83] Taha Yasseri, Robert Sumi, András Rung, András Kornai, and János Kertész. 2012. Dynamics of conflicts in Wikipedia. *PLoS one* 7, 6 (2012), e38869.
- [84] Justine Zhang, Jonathan P Chang, Cristian Danescu-Niculescu-Mizil, Lucas Dixon, Yiqing Hua, Nithum Thain, and Dario Taraborelli. 2018. Conversations gone awry: Detecting early signs of conversational failure. *arXiv preprint arXiv:1805.05345* (2018).

A INTERVIEW STRUCTURE

We conducted 22 semi-structured interviews. The duration of the interviews varied from 45 to 60 minutes. Each interview followed the structure presented below. The interviewer followed up on the respondents' narratives in the following cases: (1) to contribute to the extension of knowledge about conflicts; and (2) to understand the topics present in previous interviews or uncover new areas to explore. We paid particular attention to areas where the respondents' experience and opinions differed from other respondents.

Introduction (5 min)

- Welcome
- Information about the research and the interview
- Reminder of rights and consent with participation
- Asking for the context of work with code review

Body (35 –50 min)

- Experience with conflicts and examples
- How do the conflicts look like?
- What are the conflicts about?
- How do the conflicts influence them, their work, and their team?
- Are there any positive or negative consequences of conflicts?
- Why do the conflicts happen?
- What are the things that influence whether and how the conflict happens?
- What would help them prevent and manage conflicts?

Ending (5 –10 min)

- Space for questions
- Final remarks and summary
- Offering results and publications

B QUALITATIVE ANALYSIS SPECIFICATION

B.1 Bottom-up approach

The bottom-up approach is an inductive qualitative analysis approach where the researcher derives codes and themes from the data content, closely matching the data's content. It differs from the top-down (deductive) method in which the researcher brings in advance a series of concepts, ideas,

or topics to the analysis. However, even if the analysis is purely bottom-up, researchers may bring into the analysis at least a set of concepts and ideas that help facilitate coding [23].

Our analysis specified the topics and areas of interest based on literature and concepts related to interpersonal conflicts. The developers' narratives drove the codes themselves and the themes formed from the codes as well as the terms they have brought up in the interviews.

We chose the "bottom-up approach" during the Thematic Analysis for the following reasons: (1) Conflicts in code reviews are an emerging and poorly understood topic; (2) We conduct an exploratory study of conflicts as applied in code review specifically, thus we want to keep the analysis as close to the experience of developers as possible; (3) We aim to provide results that are beneficial to the developers, and deriving themes based on developers' narratives keeps the coding relevant to the audience; (4) We aim to mitigate the researcher's bias by preventing the researcher from bringing concepts and ideas that do not necessarily link to the semantic data content.

B.2 Peer-debriefing

Peer debriefing is a review of the research process conducted by a researcher external to the analysis yet familiar with the data or with the phenomena under investigation. The reviewer supports the analyst, challenges the researchers' assumptions, and provides critical feedback on the analysis's progress [29].

During the initial coding phase of the thematic analysis (Step 2 in Section 3.5), to discuss the coding scheme, the first author had meetings regularly (weekly or bi-weekly) with the second author, who had familiarized with the data by reading the complete set of interview transcripts. These meetings aimed to introduce the second author to the current structure of codes. The analyst (the first author) could ask clarifying questions about possible meaning and interpretation of data or software engineering practices and terminology. The reviewer (the second author) had the opportunity to ask for codes' explanation, to point out inconsistencies in the analysis, to provide observations about the data, and to relate to the current state and potential extensions of the analysis. All three authors discussed the outcome of the peer debriefings in a meeting before the third author proceeded with the audit.

B.3 Audit-like process

The audit trail is a procedure to validate the qualitative analysis results. It requires the researcher to provide detailed information on how the analysis was conducted to auditors external to the analysis. A formal audit aims to examine both the inquiry's process and product and determine the findings' trustworthiness. Typically, an external auditor, such as a thesis reviewer, conducts the audit. However, the auditor in this study was the third author. Therefore, the audit's position towards the analysis is not impartial due to the auditor's familiarity with the project. Furthermore, the third author took part in the study's methodological and writing process and discussions. Therefore, we call the audit in this study an "audit-like" process—an audit performed by a person external to the analysis yet internal to the project. An audit-like process is also among the methods used to enhance the trustworthiness of qualitative studies [20].

The audit was performed on several documents, including a text summarizing the related work, methodology, analysis results, the analyst's notes, theme definitions, the NVivo file with complete coding, and interview transcripts. The auditor read three interviews with abundant coding to familiarize themselves with at least a third of the relevant data. Then the auditor proceeded with the documents covering the analysis, results, coding, notes, and definitions of themes.

The audit aimed to reach the following goals:

- to get acquainted with the methodology and results of the analysis and related documentation

- to review whether the analysis results are a good representation of the data
- to review whether the analysis does not breach knowledge available to software engineering unless well supported
- to review whether the results are beneficial for the community
- to control for issues and shortcomings of the analysis

The auditor proposed changes and improvements to the themes' organization and reporting of the themes. However, no shortcomings in the validity of the analysis were identified in the audit.