




# Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load

Pavlına Wurzel Gonçalves<sup>1</sup> · Enrico Fregnan<sup>1</sup>  · Tobias Baum<sup>2</sup> · Kurt Schneider<sup>2</sup> · Alberto Bacchelli<sup>1</sup>

Accepted: 6 January 2022 / Published online: 7 May 2022  
© The Author(s) 2022

## Abstract

Code review is an important process in software engineering – yet, a very expensive one. Therefore, understanding code review and how to improve reviewers' performance is paramount. In the study presented in this work, we test whether providing developers with explicit reviewing strategies improves their review effectiveness and efficiency. Moreover, we verify if review guidance lowers developers' cognitive load. We employ an experimental design where professional developers have to perform three code review tasks. Participants are assigned to one of three treatments: ad hoc reviewing, checklist, and guided checklist. The guided checklist was developed to provide an explicit reviewing strategy to developers. While the checklist is a simple form of signaling (a method to reduce cognitive load), the guided checklist incorporates further methods to lower cognitive demands of the task such as segmenting and weeding. The majority of the participants are *novice reviewers* with low or no code review experience. Our results indicate that the guided checklist is a more effective aid for a simple review, while the checklist supports reviewers' efficiency and effectiveness in a complex task. However, we did not identify a strong relationship between the guidance provided and code review performance. The checklist has the potential to lower developers' cognitive load, but higher cognitive load led to better performance possibly due to the generally low effectiveness and efficiency of the study participants. Data and materials: <https://doi.org/10.5281/zenodo.5653341>. Registered report: <https://doi.org/10.17605/OSF.IO/5FPTJ>.

**Keywords** Code review · Reviewing strategies · Checklist-based reading · Cognitive load

---

Communicated by: Neil Ernst

This article belongs to the Topical Collection: *Registered Reports*

✉ Pavlına Wurzel Gonçalves  
p.goncalves@ifi.uzh.ch

Extended author information available on the last page of the article.

## 1 Introduction

Code review is a widely used (Rigby and Bird 2013; Bacchelli and Bird 2013; Gousios et al. 2014; Sadowski et al. 2018) software engineering practice in which one or more reviewers inspect a code change written by a peer (Bacchelli and Bird 2013; MacLeod et al. 2017) to improve software quality (Baum et al. 2017a), find defects (Baum and Schneider 2016), and transfer knowledge (Bacchelli and Bird 2013).

Performing efficient and useful code reviews is an expensive and time-consuming task (Cohen 2010), therefore improving developers' performance during code review is of great interest. Performance in the context of code review is often defined as how many defects are found (*effectiveness*) in the code change under review and in how much time (*efficiency*) (Biffi 2000).

The mentally challenging nature of reviewing code is one of the reasons why code review is expensive (Pascarella et al. 2018; Bacchelli and Bird 2013; Baum 2019). To find defects, developers need to process a vast amount of information related to the code change, to its rationale, to its context in the whole codebase, and to its implications for software quality (Pascarella et al. 2018). Understanding a change-set to review (*e.g.*, a pull request (Gousios et al. 2014)) and its context is one of the main challenges of code review (Tao et al. 2012; Bacchelli and Bird 2013).

The cognitive resources (*e.g.*, working memory capacity) are available to developers during code review can impact review performance. For example, working memory capacity is helpful to find *delocalized* (Dunsmore et al. 2003) defects (*i.e.*, defects that can only be identified by inspecting non-contiguous parts in a program) (Baum et al. 2019). However, working memory is a limited resource (Paas et al. 2003) and the cognitive load that a task poses on the cognitive system can deplete the available capacity, thus leading to cognitive overload and poor performance (Paas et al. 2003; Matthews et al. 2019).

In recent years, researchers devised approaches to support the code review process, such as visualizations (Tymchuk et al. 2015; Oosterwaal et al. 2016), optimizations of the order in which review files are displayed (Baum et al. 2017b; Baum 2019), and untangling of unrelated changes in a changeset under review (Barnett et al. 2015; Dias et al. 2015; Tao and Kim 2015). These aim at increasing developers' review performance. Although most of these approaches do not directly aim to reduce developers' cognitive load, they do improve reviewers' ability to understand the change-set under review and navigate it—activities that require high cognitive resources.

Existing tools to support developers and improve their code review performance do not guide developers on *how* to perform the review, even though this kind of guidance could help to lower required cognitive resources (Mayer and Moreno 2003). Rather, to give this kind of support to reviewers, researchers investigated reading techniques for formal code inspection (Fagan 2002; Basili et al. 1996). These techniques guide developers in *how* to inspect the code searching for defects (Baum 2019).<sup>1</sup>

A reading technique for code review used in industry (Baum 2019; Gutha 2015; Gridnev 2017; Carver 2003) is checklist-based reading (Fagan 2002). A checklist guides developers in what and how to review by providing explicit instructions: For instance, a checklist might ask developers to “check an issue for each method” (Kamsties and Lott 1995). Checklists explicitly aim to aid developers in performing complex tasks by systematizing their

---

<sup>1</sup>Tools that follow a similar principle have been developed (*e.g.*, (LaToza et al. 2020)), but to support other software development tasks, such as testing and debugging. The tool developed by LaToza et al. (2020) provides means to follow and execute explicit strategies on how to perform the task.

activity, thus lowering the cognitive load of the task (Kamsties and Lott 1995; LaToza et al. 2020). However, the relationship between checklists and lowered cognitive load has yet to be empirically tested. Moreover, checklists provide a basis to develop an executable reviewing strategy: Automatizing the flow of its items, a checklist can be turned into a step-by-step strategy.

In this paper, we present a study we designed to explore how to assist developers in decreasing the complexity and cognitive challenges of code review, focusing specifically on checklist-based code review. Our aim is to test whether implementing a reviewing strategy using additional methods of cognitive load reduction in a code review tool leads to improved review effectiveness and efficiency. We use *checklists* as a code reading technique and we aim to improve the code review performance by (1) providing the steps on how to execute the review, (2) strengthening the tool-support to systematically execute the checklist, and (3) making the review more focused and flexible to fit better the change-based characteristics of modern code review. To this aim, we developed a guided checklist, whose step-by-step execution is supported by a tool and reflects the content of the specific review. We measure how this method compares to a normal checklist and a control group. Particularly, we investigate whether these approaches improve performance through lowering cognitive load and assess the usability of the implemented guidance approaches.

The research design of this study was accepted as a Registered Report at MSR'20 (Gonçalves et al. 2020). Accordingly, we conducted an experiment with 70 developers who performed three review tasks. The experiment has three treatments: (1) ad hoc reviewing,<sup>2</sup> (2) checklist-based reviewing, and (3) guided checklist-based reviewing (which uses further means of reducing cognitive load). After each review, we measured developers' cognitive load.

The majority (71.6%) of the developers who eventually took part in our experiment does not commonly practice code review—they can be considered as *novice reviewers*.

The participants achieved low review effectiveness and efficiency regardless of the treatment to which they were assigned, therefore limiting the strength of our results. Nevertheless, we provide an initial indication on the relationships between guidance, code review performance, and cognitive load. Our results show that the guided checklist performs better in a simpler task: Using a regression model, we identified a statistically significant relationship between the use of the guided checklist and review effectiveness in the small review task (*Small Change*). The checklist, instead, seems to increase our participants' review effectiveness and efficiency in the more complex tasks: We identified the existence of a relationship between the use of the checklist and higher review effectiveness and efficiency in one of the large review tasks (*Large Change B*).

Moreover, we observed that a higher cognitive load is linked to better performance. This contradicts our expectations. This result might have been caused by the generally low review performance of the participants and could indicate that investing cognitive resources is actually needed to perform well for novice reviewers.

Registered report: <https://doi.org/10.17605/OSF.IO/5FPTJ>.

## 2 Background and Related Work

Over the years, substantial research has been dedicated to improving developers' performance during peer code review. Some approaches focus on giving developers information

---

<sup>2</sup>This treatment provides no guidance during the review.

on the context of a review change-set: *e.g.*, employing visualizations to show the structure of the code (Tymchuk et al. 2015) and finding potential issues with the change, based on similar changes in the codebase (Zhang et al. 2015). Other approaches focus on simplifying complex review change-sets by decomposing them into groups of related changes (Tao and Kim 2015; Barnett et al. 2015; Dias et al. 2015).

Code review is a cognitively demanding task (Baum 2019; Pascarella et al. 2018; Bacchelli and Bird 2013). For this reason, researchers devised approaches to lower reviewers' cognitive load during code review. For instance, Baum et al. (2017b) proposed to order review changes based on their relations (instead of using the alphabetical order of the file names as done by popular code review tools, such as Gerrit and GitHub), as a way to lower the effort developers need to put in understanding of the construction, connections, and logic of the changes to review.

In the following section, we expand on the role that cognitive load plays during code review and present how current tools help to reduce reviewers' cognitive load.

## 2.1 Cognitive Load and Reviews

Working memory is the part of human memory in charge of storing short-term information in processing tasks. It remains stable throughout a person's life and cannot be significantly trained or improved (Dobbs and Rule 1989). Research found evidence that working memory capacity is linked to the capacity of finding delocalized defects during code review (Baum et al. 2019). In fact, finding delocalized defects requires simultaneous cognitive processing of different parts of the code.

Cognitive load refers to the amount of working memory used while performing a task (Paas et al. 2003). Once the cognitive load exceeds one's working memory capacity, their performance in the task lowers considerably (Matthews et al. 2019). More difficult tasks (*e.g.*, more challenging code reviews) pose a higher cognitive load and deplete working memory capacity faster. Supporting people in using less working memory capacity while performing their tasks can prevent them from reaching working memory overload. Moreover, this kind of support might also help those with lower working memory capacity to perform well in complex tasks (Bannert 2002).

When it comes to processing information, there are three types of cognitive load at play that contribute to the total cognitive load and potential overload. Since it is important that the cumulative cognitive load does not exceed the working memory capacity (Paas et al. 2003), the goal should be to minimize the cognitive load caused by processing the information related to efficiently solving a task (intrinsic and extraneous load) and free capacity for the load used for dedicated and focused performance (germane load) (Bannert 2002).

**Intrinsic load:** The *intrinsic* load relates to the complexity of a task. It refers to the amount of interacting elements that must be simultaneously handled by the working memory. The intrinsic load can be lowered by simplifying the task or reducing the amount of interacting elements. The human mind can deal with intrinsic cognitive load by storing information in the long-term memory and retrieving it only when needed or by automating repeated cognitive processes and behaviors. For this reason, experience is fundamental to reach efficiency in a task (van Bruggen et al. 2002). Tools that contribute to lowering the intrinsic load in software development help with these functions – by storing information and providing it in the right moment (LaToza et al. 2020) or by automating repetitive tasks (Rafi et al. 2012). Some tools to support code review simplify unnecessary processing of interacting elements by partitioning changes into smaller

related portions (Tao and Kim 2015; Barnett et al. 2015) or by providing a summary through visualizations (Tymchuk et al. 2015).

**Extraneous load:** The *extraneous (ineffective)* load is caused by the need to process unnecessary or unrelated information, thus harming the performance. For instance, the need to switch contexts/documents, to understand unclear documentation, and to search for information without available pointers are situations impacting the extraneous cognitive load.

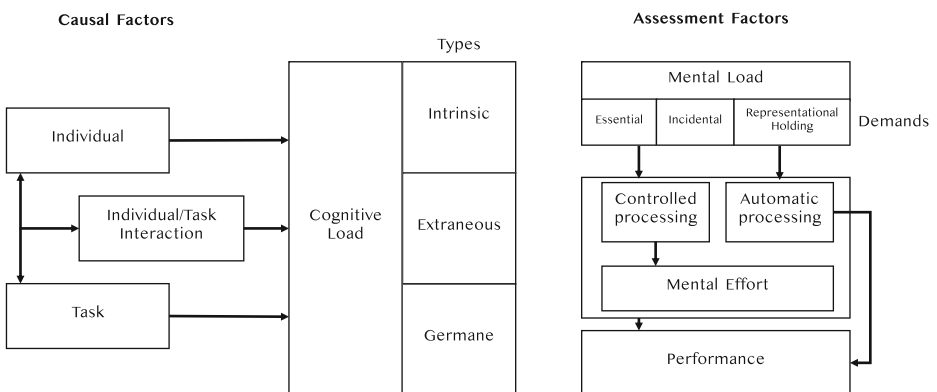
Checklists can also be employed to reduce reviewers’ extraneous cognitive load. They help developers to focus their attention on the specific areas of code that need inspection (Gutha 2015; Gridnev 2017; Rong et al. 2012).

**Germane load:** The *germane (effective)* load comes from the effort put in solving a task. This type of load is helpful for developers’ effectiveness and efficiency. It is related to motivation (higher determination also poses higher germane load) but also to previous knowledge about the issue (less effort is needed to solve the task if the developer already has the needed knowledge). In practice, this type of cognitive load can be created, for instance, by introducing gamification in code reviews to improve the interest and motivation of developers (Khandelwal et al. 2017; Unkelos-Shpigel and Hadar 2015).

Intrinsic cognitive load is the most difficult to manipulate, as often there is no choice in how complex the materials necessary for completing a task are. This limits the working memory available to deal with sub-optimal information inputs or that can be put in motivated and dedicated performance. Therefore, the effect of lowering the *ineffective load* and raising the *effective* one is particularly important when dealing with challenging reviews (Bannert 2002).

As shown in Fig. 1, the cognitive load for an individual is determined by their characteristics (e.g., available knowledge), the characteristics of the task (e.g., complexity, type of problem), and their interaction (experience with a specific type of problem). Therefore, to lower the cognitive load in a task, interventions can be done to improve individuals’ abilities, adjust tasks to pose a lower cognitive load, or optimize the fit between the needs and possibilities of individuals and the tasks they are performing.

With respect to measuring and assessing cognitive load is conceptualized through the affected factors represented on the right-hand side in Fig. 1 (Paas and Van Merriënboer 1994). *Mental load* represents the demands posed on the cognitive system by the task itself, while *mental effort* represents the cognitive demands consciously allocated to solving the task.



**Fig. 1** Conceptual framework for understanding and assessing cognitive load, adapted from Paas and Van Merriënboer (1994)

Several methods have been proposed to lower mental load, *i.e.*, the cognitive demands of tasks. Each of them addresses different types of cognitive demands. Apart from the demands originated from the complexity of the information (*essential* to the task) and the demands caused by processing the *incidental* (unessential) information, it is demanding to *hold* the information in working memory for a long time (Mayer and Moreno 2003).

Table 1 presents a list of methods to reduce mental load from Mayer and Moreno (2003). Among these methods, some have already been integrated into tools to support code review. For instance, tool support such as visualizations or change ordering incorporate methods such as parallel processing of verbal and graphical information, reducing visual scanning, segmenting information, or reducing the processing of unessential information for the review.

In summary, mental load and mental effort both contribute to the overall cognitive load and potential cognitive overload. Making review less demanding on available cognitive resources can help to prevent cognitive overload. Therefore, in this study we aim to prevent cognitive overload by reducing the mental load code review is posing on the developers' mind. We measure cognitive load as the goal concept and refer only to this concept in the following sections.

## 2.2 Code Inspection Reading Techniques and Strategic Reviewing

This study aims to explore whether the use of an explicit strategy to review code improves review performance. The idea of defined processes and steps for reviewing is integral to formal code inspection (Ebad 2017).

Over the years, multiple reading techniques have been developed to guide developers in inspecting code and other types of documents (Ebad 2017). Some code reading techniques are simple checklists that focus reviewers on certain aspects to ensure that these are checked, while others offer an explicit step-by-step guide to follow to review the artifact at hand (Baum 2019). However, modern code review does not commonly apply formal inspection reading techniques (Baum et al. 2017a), focusing more on the advantages offered by the use of review-specific tools (*e.g.*, Gerrit<sup>3</sup>, Microsoft CodeFlow (Greiler 2021), Facebook's Phabricator<sup>4</sup>, and Atlassian Crucible<sup>5</sup>), as also described in Section 2.1.

Checklists are an example of a reading technique that has been used not only for code inspection (Thelin et al. 2003), but also for other types of code review (Gutha 2015; Gridnev 2017; Rong et al. 2012). Checklists utilize *signaling* (see Table 1) and are thought to improve performance through lowering cognitive load (Kamsties and Lott 1995). They have been found to be an efficient aid for finding defects (Rong et al. 2012), but are outperformed by reading techniques that follow a specific reviewing scenario (Abdelnabi et al. 2004; Denger et al. 2004). This suggests that guidance that shows reviewers how to proceed with the review by further *signaling* cues for what to look for, where, and when may be beneficial. Nevertheless, the positive effect of explicit strategies is not supported by all studies (McMeekin et al. 2009; Lanubile et al. 2004) and checklists seem to be better accepted by reviewers compared to reading scenarios (Lanubile et al. 2004).

The importance of defined cognitive processes and their systematic execution is recognized when aiding software development tasks like debugging (LaToza et al. 2020; Ko et al.

<sup>3</sup>Gerrit Code Review: <https://www.gerritcodereview.com>

<sup>4</sup>Phabricator: <https://secure.phabricator.com>

<sup>5</sup>Atlassian Crucible: <https://www.atlassian.com/software/crucible>

**Table 1** Mental Load Reducing Methods (from (Mayer and Moreno 2003)), Code Review Tools and Experimental Treatments

Definition	Cognitive Demands Addressed	Code Review Tool Support	Treatment: Checklist	Treatment: Guided Checklist
<i>Off-Loading</i> Allow parallel processing of verbal and graphical information.	Essential	Visualizations (Tymchuk et al. 2015)	.	.
<i>Segmenting</i> Present information in segments rather than as a continuous unit.	Essential	Partitioning Changes (Tao and Kim 2015)	.	x
<i>Pre-training</i> Provide knowledge of names and behaviors of system components.	Essential	Documentation (Ciurumelea et al. 2020), Review suggestions (Zhang et al. 2015)	.	.
<i>Weeding</i> Eliminate or reduce processing of unessential information for the task.	Essential + Incidental	Change Ordering (Baum et al. 2017b), Untangling (Dias et al. 2015) and Decomposing (Barnett et al. 2015), Automation (Rafi et al. 2012)	.	x
<i>Signaling</i> Provide cues for how to process the material.	Essential + Incidental	Review suggestions (Zhang et al. 2015)	x	x
<i>Aligning</i> Place related information together to reduce visual scanning.	Essential + Incidental	Visualizations (Tymchuk et al. 2015), Change Ordering (Baum et al. 2017b), Review suggestions (Zhang et al. 2015)	.	x
<i>Eliminating Redundancy</i> Avoid unnecessary repeating of information.	Essential + Incidental	Review suggestions (Zhang et al. 2015)	.	.
<i>Synchronizing</i> Present verbal and graphical information simultaneously to minimize the need to hold information in working memory.	Essential + Representational Holding	Visualisations (Tymchuk et al. 2015)	.	x
<i>Individualizing</i> Take into account individual resources and abilities, such as working memory or learning style.	Essential + Representational Holding		.	.

2019). These strategies for executing programming tasks take advantage of the functionalities that tools provide to lower developers' cognitive load by storing and managing the information needed to solve the issues.

By building on methods to reduce mental load, we have developed a tool-supported reviewing strategy to assist developers in improving their code review performance.

### 3 Research Questions

Supporting developers with reading techniques (*e.g.*, checklist) has been found to be an efficient way to help reviewers find defects during code inspection (Biffel 2000). However, checklists were found to be less effective compared to reading techniques that offer guidance on how to review, such as Systematic Order-based Reading (Abdelnabi et al. 2004). Assisting developers in defining and executing strategies for software development tasks (*e.g.*, debugging) increases developers' productivity (LaToza et al. 2020).

The positive outcome of previous research suggests that incorporating methods to reduce developers' cognitive load (*e.g.*, signaling (Mayer and Moreno 2003)) can positively affect review performance. In this study, we investigate whether code review efficiency and effectiveness can be improved using additional methods to reduce developers' cognitive load. We compare a tool-supported systematic guidance on *how* to perform a review (guided checklist) to guidance only on *what* to look for in the review (checklist) and to an ad hoc review where developers perform the review according to their own process. Our first research question is the following:

**RQ1:** Does guidance in review lead to:

**RQ1.1:** a higher review effectiveness (proportion of functional defects found)?

**RQ1.2:** a higher review efficiency (functional defects found over the review time)?

We formalize our research question into the following hypotheses:

**H<sub>1,1</sub>:** There are differences in review effectiveness between ad hoc review, checklist, and guided checklist.

**H<sub>0,1,1</sub>:** There are **no** differences in review effectiveness between ad hoc review, checklist, and guided checklist.

**H<sub>1,2</sub>:** There are differences in review efficiency between ad hoc review, checklist, and guided checklist.

**H<sub>0,1,2</sub>:** There are **no** differences in review efficiency between ad hoc review, checklist, and guided checklist.

Checklists, as well as tool-supported strategies, are expected to systematize the activity of the developers, thus lowering their cognitive load by reducing the amount of information they have to keep in mind and helping them to focus on relevant issues (Paas and Van Merriënboer 1994; LaToza et al. 2020). In the software engineering literature, however, we found no direct measurement of the effect of tools on cognitive load and its effect on code review performance. Therefore, we ask:



**RQ2:** Is the effect of guidance:

**RQ2.1:** on review effectiveness mediated by a lower cognitive load?

**RQ2.2:** on review efficiency mediated by a lower cognitive load?

We formalize our research question as:

**H<sub>2.1</sub>:** Cognitive load mediates the relationship between the guidance approach and review effectiveness.

**H<sub>0.1</sub>:** Cognitive load does not mediate the relationship between the guidance approach and review effectiveness.

**H<sub>2.2</sub>:** Cognitive load mediates the relationship between the guidance approach and review efficiency.

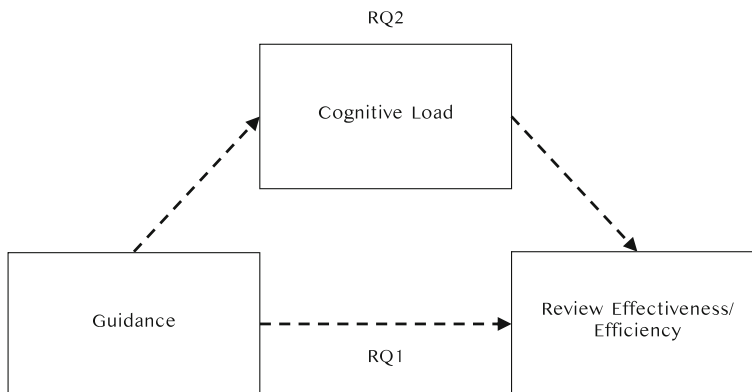
**H<sub>0.2</sub>:** Cognitive load does not mediate the relationship between the guidance approach and review efficiency.

Figure 2 summarizes our research questions and their link with the key concepts of our investigation: *e.g.*, cognitive load and review performance.

## 4 Methodology

After having reported the goal of our experiment and our hypothesis in the previous section, in this section we describe the experiment planning, following the reporting guidelines of Jedlitschka et al. (2008).

According to the methodology presented in our registered report (Gonçalves et al. 2020), we set up a controlled experiment where developers have to complete three code review tasks searching for defects. Each participant is randomly assigned to one of three possible treatments: (1) a control treatment with no guidance (henceforth: ‘ad hoc review’), (2) checklist supported review (‘checklist’), and (3) strategic checklist execution (‘guided checklist’).



**Fig. 2** The research questions as related to the main theoretical concepts

## 4.1 Study Participants

As reported in our registered report (Section 3.4) (Gonçalves et al. 2020), we performed a power analysis to estimate the sample size needed to identify existing differences between the treatment groups. Based on previous studies, we do not expect a large effect size to appear (Dunsmore et al. 2001). The sample size is calculated using a convention for an ANOVA medium effect size (Cohen 1992). The estimated total sample size is 66 participants. Based on this analysis, we hired 70 developers from a software development outsourcing company located in India to take part in our experiment. The company has more than 2,000 employees and provides a wide range of services (e.g., DevOps, web development, and mobile development).

We contacted the developers through the company and they completed the experiment as part of their job. We requested all developers to have experience in Java, but we had no further control over the selection of the sample, which was up to the project manager at the company. We had the option to ask for additional developers in case of irregularities in data or drop-outs.

### 4.1.1 Descriptives

Characteristics of the study sample are described in Figs. 3, 4, and Table 6. After the data cleaning, our participants' sample consisted of 67 professional Java developers, counting 66 programmers and one tester. Among them, 54 identified themselves as male and 13 as female. The age of the participants ranged between 22 and 33 ( $M = 26.85$ ). Furthermore, we know that 28 participants had a B.Sc. in Computer science and 18 had a M.Sc. degree in Computer Science, totaling 68.7% of the study participants with a university degree.

Most participants had no experience with jEdit – the system used in the review tasks. However, five of them used it in the past and six have contributed to the jEdit code base. While analyzing collinearity in the data, we did not find a significant relationship between experience with jEdit and performance in the experiment reviews.

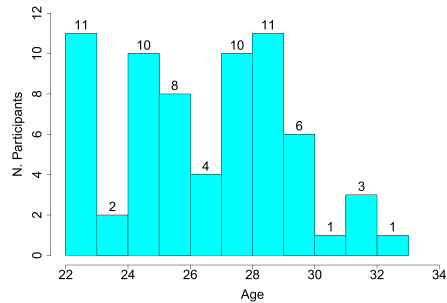
Our sample consisted of professional Java developers, however many of them did not have code review experience. Several (39) of them already worked more than 8 hours before doing the experiment and 30 of them reported being moderately or very stressed before the experiment. Participants reached low effectiveness and efficiency, as reported in Table 6. Furthermore, we verified developers' understanding of the change-set at the end of each review task. The results are shown in Table 2.

## 4.2 Experiment Treatments and Materials

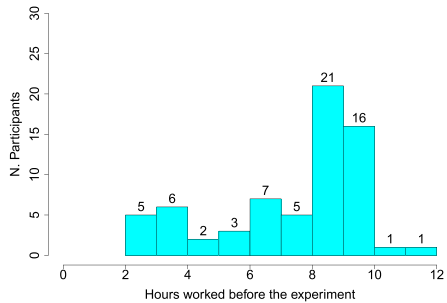
In this section, we describe the materials used in our experiment. First, we describe the experiment UI; then we present the three treatments (*i.e.*, ad hoc review, checklist, and guided checklist). Moreover, we describe how we measured the cognitive load of the participants as well as how we assessed the usability of the devised guidance approaches. An explanation of the materials is provided in our registered report (Gonçalves et al. 2020).

### 4.2.1 Experiment UI

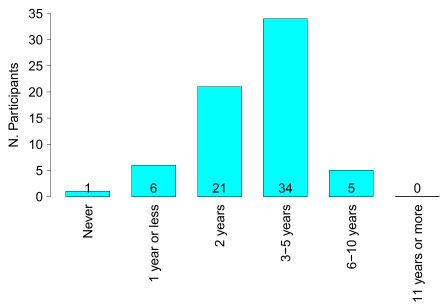
To conduct our experiment, we use a web-based tool (Fig. 5 shows an example view) that allows participants to complete the experiment remotely. We log participants' answers, environment, and UI interactions. The tool was built and used in our previous work (Baum et al.



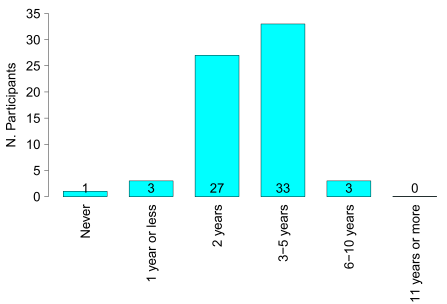
(a) Developers' age.



(b) Hours worked before participating in the experiment.



(c) Experience of developing software in a professional setting.



(d) Years of experience with Java.

Fig. 3 Participants' demographics (1)

2019) and modified according to the new experiment's requirements. The tool is available in our replication package<sup>6</sup>.

### 4.2.2 Treatment: Ad hoc Review

The *ad hoc review* (Uwano et al. 2006) condition is our control group, which we use as a baseline to evaluate participants' review performance. Developers assigned to this treatment use the same web-based experiment platform as the other treatments to perform the review tasks. All participants review the same tasks regardless of the treatment to which they are assigned. Developers in the ad hoc review group do not receive any specific aid during the review and can carry on the reviews as they prefer.

### 4.2.3 Treatment: Checklist

Checklists provide cues on where to focus attention to find common defects and improve the usage of cognitive resources (Bannert 2002). This can be seen as using *Signaling* to reduce cognitive load (see Table 1).

<sup>6</sup>Replication package: <https://doi.org/10.5281/zenodo.5653341>

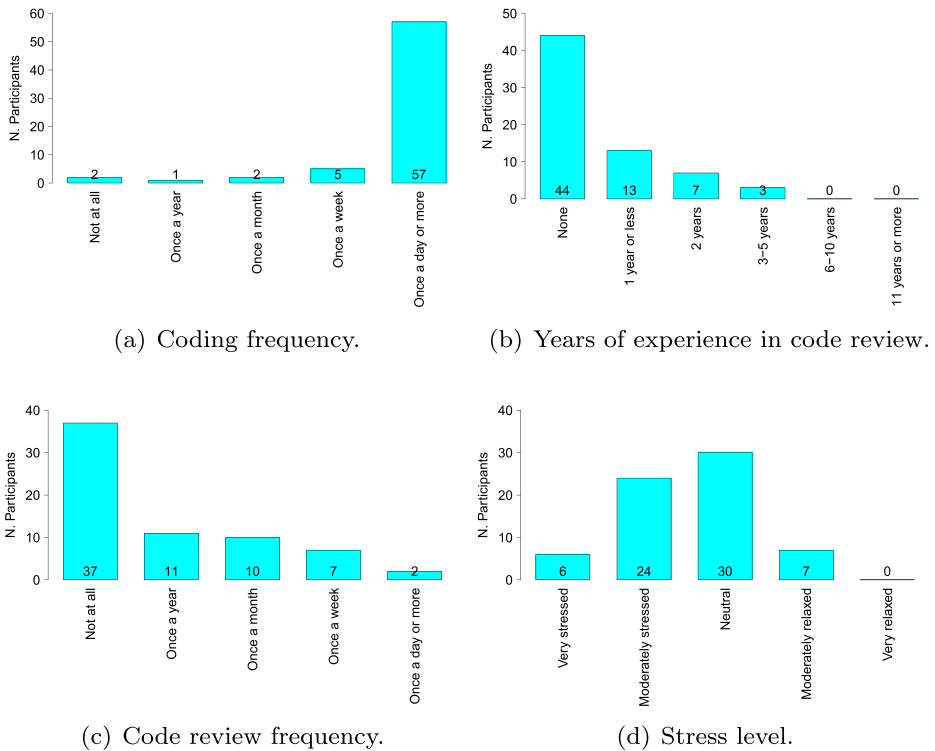


Fig. 4 Participants’ demographics (2)

Developers assigned to the “Checklist” treatment of our experiment are required to identify defects using a checklist. We developed this checklist based on items from Microsoft checklists (McConnell 2004) and recommendations in the literature: A good checklist (1) requires a specific answer for each item, (2) separates items by topic, and (3) focuses on relevant issues (Degani and Wiener 1991; Kamsties and Lott 1995; Chernak 1996). Checklists should specify the scope in which items must be checked (*e.g.*, “for each method/class”) to prevent developers from memorizing big portions of code and jumping through it (Kamsties and Lott 1995).

Following these recommendations, we created the checklist for our experiment. For each defect, our checklist contains at least one item related to the issue but without explicitly pointing to it. Thus, the checklist contains items relevant to the review at hand but does not give obvious clues about the type or location of the defects.

Table 2 Participants’ scores in the understanding questions shown at the end of each review task

Change	Mean	Standard deviation	Num. of understanding questions
Small/Warm-up	1.21	0.71	2
Large Change A	1.45	0.86	4
Large Change B	1.37	0.76	3



Fig. 5 Partial view of the checklist implementation in the web-based experiment UI

Every item in the checklist was extracted from the Microsoft checklists presented by McConnell (2004). We created an initial version of the checklist and performed an assessment with three Java developers with experience in code review to evaluate its goodness. Based on the collected feedback, we improved the items in our checklist. Then, we repeated this process with a new set of three developers.

The final version of the checklist contains 18 items, grouped by their scope (general, class, or method). For each item, developers can indicate whether they considered it without finding any defect or they found a defect while inspecting it. Reviewers are not forced to check every item, but a warning is shown if they attempt to complete the review without having marked all checklist items as checked. The checklist is displayed as a lateral bar on the left side of the screen (Fig. 5). Developers can open or close the checklist bar by clicking on the *collapse checklist* button on the top-left corner of the screen. Furthermore, we ask developers to note any defects that they encounter, even if they are unrelated to the content of the checklist.

The checklist items are reported in Appendix A.1; the mapping to the defects they help identify is in our replication package<sup>7</sup>.

#### 4.2.4 Treatment: Guided Checklist

The human brain has a great potential to retrieve complex information and consequently make contextualized decisions. A tool-supported strategy could free the mental capacity to do these tasks by aiding systematic execution of steps and providing relevant information when needed (LaToza et al. 2020). Therefore, providing explicit strategies to perform code review might support developers by reducing their cognitive load and improving their performance.

The guided checklist is a version of the previous checklist (Section 4.2.3). Checklists should specify the scope for which an item must be checked: e.g., “for each class”. Differently from a classic checklist, the guided checklist is not static but iterates over the classes

<sup>7</sup>Replication package: <https://doi.org/10.5281/zenodo.5653341>

and methods in the review change-set. This allows a detailed step-by-step review of each relevant checklist item, e.g., “For the class *VFSBrowser*, please check . . .”.

In comparison to the checklist, the implementation of the guided checklist is improved by multiple methods for lowering cognitive load, as seen in Table 1. The guided checklist uses the same items and *signals* what to look for. Additionally, it (1) segments the task into smaller units, (2) reduces the need to hold information in the working memory by iterating through classes and methods, (3) reduces visual scanning by highlighting chunks and asking focused questions on a specified piece of code, and (4) minimizes the processing of unessential information by offering only items relevant to that chunk. Therefore, even though both checklist and guided checklist use signaling as a method to reduce cognitive load, the guided checklist is expected to reduce the amount of information a developer needs to process at a time and the scope to which they need to pay attention—the signal is more precise. Thanks to the identical content of checklist and guided checklist items, we can conceptually separate the effect of additional measures for reducing cognitive load.

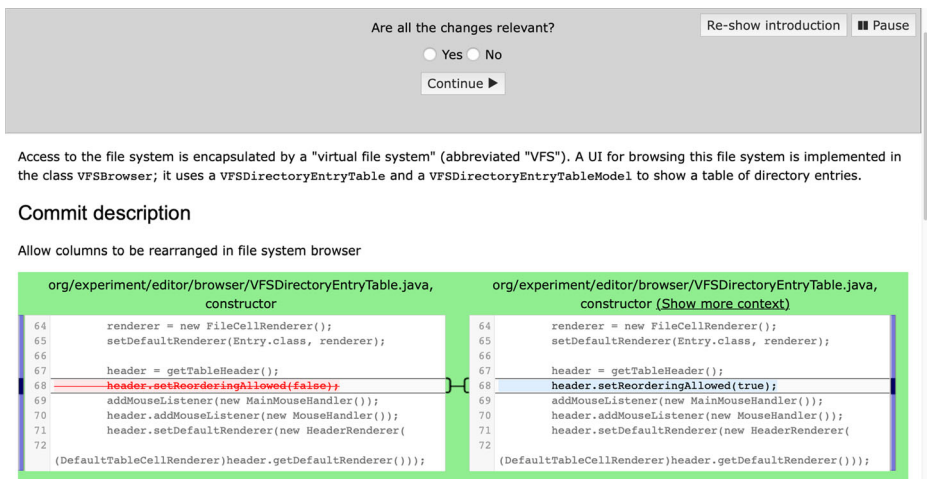
---

**Algorithm 1** Guided checklist: Execution flow

---

- 1: Familiarize with the review change-set
  - 2: Check the overall relevance of the changes regarding the requirements
  - 3: **for** All classes in review change-set **do**
  - 4:     Check all checklist items relevant for the class
  - 5:     **for** All methods in a class **do**
  - 6:         Check all checklist items relevant for the method
  - 7: Final check on changes’ consistency
- 

The guided checklist is implemented as a top bar in the review task interface (Fig. 6). It displays the same items as the checklist. Differently from the checklist, items are not shown all at the same time, but participants are explicitly asked first to check the general items, then the class and method ones. The execution flow of the guided checklist is reported in Algorithm 1. We display only the items that are relevant for the selected code chunk. Furthermore, the strategy highlights to the user which code chunk(s) they are currently



**Fig. 6** Example of a strategy item in the web-based experiment UI

reviewing. The user must explicitly mark the items as checked before being able to proceed with the review.

#### 4.2.5 Cognitive Load

To measure cognitive load, we use a standardized questionnaire (StuMMBE-Q (Krell 2017)) that captures the two components of cognitive load (*i.e.*, mental load and mental effort) in two 6-item sub-scales. The items are rated on a 7-point Likert scale. The individual responses are recorded as a score from 1 to 7. These scores are averaged to achieve a final score directly comparable to the response anchors. The scale contains no reverse-scored items. Effort and difficulty ratings are reliable measures for the cognitive processing that contributes to cognitive load (DeLeeuw and Mayer 2008). While there are other potential measures of cognitive load, such as response time to a secondary task (Paas et al. 2003), we use a questionnaire because it does not require the physical presence of the respondents or the usage of special equipment. Moreover, it does not directly interfere with the code review performance.

#### 4.2.6 Usability of the Treatment Implementation

We adapted the System Usability Scale (Brooke and et al 1996) to measure the usability of the devised guidance approaches: We rephrased the items of the System Usability Scale to fit the purpose of the checklist and guided checklist evaluation. Using the scoring manual, the treatment is graded on the scale from A to F, Excellent to Awful (UsabiliTEST 2020).

### 4.3 Tasks

Participants in our experiment were asked to complete three code review tasks. Moreover, before starting the review, developers were shown a tutorial to familiarize themselves with the review UI used in the experiment. In the following, we describe how the tutorial and the review tasks are implemented. The code of the tasks used in the experiment is available in our replication package<sup>8</sup>.

#### 4.3.1 Tutorial

The tutorial (Fig. 7) shows a brief code review consisting of one file. Reviewers are asked to perform either three tasks (in the ‘checklist’ or ‘guided checklist’ conditions) or two tasks (in the ‘ad hoc’ condition) to familiarize themselves with the review UI before proceeding to the experiment. (1) click on the *view more context* button to expand the context of a review change; (2) insert a remark; (3) if developers were assigned to the checklist or guided checklist treatments, mark an item of the checklist (guided checklist) as complete. The code to be reviewed contains a bug<sup>9</sup> for the reviewers to find.

#### 4.3.2 Code Review Tasks

First, participants review a short, simpler change-set, then they have to do two reviews of two distinct, longer change-sets. The first change-set contains three defects, while the others

<sup>8</sup>Replication package: <https://doi.org/10.5281/zenodo.5653341>

<sup>9</sup>A plus sign was replaced with a minus in a function that sums two numbers (passed as parameters).

## Review task tutorial

This is a brief tutorial to introduce how the review tasks of this experiment are conducted.

### Code changes

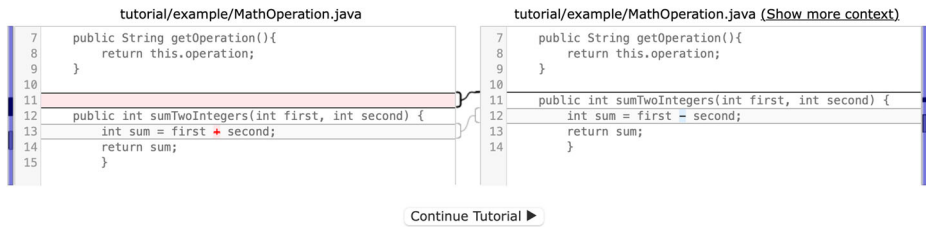
Below you find the code changes to review. The old version of the code is on the left, the new version is on the right.

In several of the change parts, you can show the whole changed method by clicking on "(Show more context)".

### Tutorial instructions

To proceed in the tutorial please complete the following instructions:

- Click on ("Show more context") and then on the continue button at the bottom of the page.



**Fig. 7** The Experiment Tutorial. In the first task of the tutorial, participants need to click on the *show more context* button to expand the context of a review change, and then click on the *continue* button to proceed to the next task of the tutorial

contain nine and ten, respectively. The two large review tasks are presented to the participants in a randomized order. Among the initial 70 participants, 36 reviewed change *Large Change A* first, while 34 were assigned to change *Large Change B* first. Table 3 describes the review tasks (also available in our replication package) and Table 4 provides the number of participants assigned to each combination of treatment (ad hoc review, checklist, or guided checklist) and change order (*Large Change A* first or *Large Change B* first).

The code changes for the review are taken from a previous experiment on code review and contain both original and seeded defects (Baum et al. 2019). The review changes are extracted from an existing open-source project named `jEdit`<sup>10</sup> that was successfully employed in previous studies (Rothlisberger et al. 2012; Baum et al. 2019). To control for potential bias caused by developers' familiarity with this system, we explicitly ask developers about their previous experience with it. We instruct the participants to focus only on functional defects.

## 4.4 Variables

Our study relies on a number of quantitative measures concerning both the performance and the perception of the participants. Table 5 reports the variables considered in our study and presented in Section 3.1 of our registered report (Gonçalves et al. 2020).

### 4.4.1 Remark Evaluation and Review Performance

Developers enter their review comments (*remarks*) in the code review UI by writing in a text area that appears once they click on a code line. As done in previous experiments (Baum

<sup>10</sup><http://www.jedit.org>



**Table 3** Code change sizes, complexity, and number of correctness defects. The code and defects were previously used and described in Baum et al. (2019)

Change	Changed Files	Change Parts	Presented LOC (a)	Cyclomatic Complexity (b)	Total Defects
Small/Warm-up	2	3	31	12	3
Large Change A	7	15	490	57	9
Large Change B	7	21	233	83	10

<sup>(a)</sup>The amount of lines of code visible to the participant on the right (=new) side of the two-pane diffs without expanding the visible context

<sup>(b)</sup>total cyclomatic complexity (McCabe 1976) of the methods on the right (=new) side of the two-pane diffs

et al. 2019), we count a comment as referring to a defect *iff* it is in the right position and can make a reader aware of the defect. In case a comment is on the right line but highlights an unrelated issue, we do not count it.

The first two authors independently classified the comments of the ten developers assigned to the first iteration of the experiment. The first iteration contained 131 remarks. These were marked as either pointing to a defect (specifying which defect) or as *false positives*. Then, we computed the agreement between the two authors involved in this process using Cohen's kappa (Kvålseth 1989). They reached an inter-rater agreement of 0.769; disagreements ( $N = 7$ ) were discussed to reach a consensus.

Afterward, we proceeded in an iterative fashion: The two authors independently evaluated two other batches of 131 remarks each. Then, the authors computed the agreement and discussed cases of disagreement until a consensus was reached. A Cohen's kappa of 0.891 and 0.806 was reached classifying the second and third batches of remarks, respectively. Since the inter-rater agreement between the authors involved in the classification achieved good results, the rest of the remarks were split between the authors for the classification. At this stage, the authors discussed only cases deemed as unclear during their individual work ( $N=14$ ).

Once all the remarks are classified, we evaluate the review performance (review effectiveness and efficiency) of the experiment participants. We measure (1) developers' review effectiveness (the percentage of discovered defects in the task) and (2) review efficiency (the number of defects found per minute spent reviewing).

#### 4.5 Experiment Design and Procedure

In our experiment, we use the *measurement-of-mediation design* (Spencer et al. 2005). The experimental design manipulates the independent variable (type of guidance), while the

**Table 4** Number of initial participants ( $N=70$ ), assigned to each combination of guidance approach (ad hoc review, checklist, guided checklist) and review change order (Large Change A first or Large Change B first)

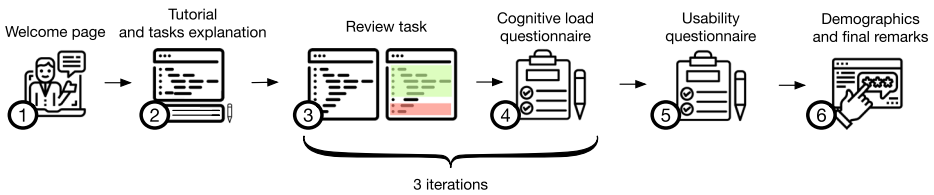
Treatment	Change A First	Change B First	Total
Ad hoc	14	16	30
Checklist	13	11	24
Guided checklist	9	7	16

**Table 5** The final variables of the study. The original table is reported in our registered report (Gonçalves et al. 2020)

Name	Description	Scale	Operationalization
<i>Independent variables (design):</i>			
Guidance approach	The guidance provided by the tool to the participant: none, checklist, or guided checklist	nominal	see Section 4.2
<i>Dependent variables:</i>			
Number of detected defects	Total functional defects found by the participant over all reviews	ratio	see Section 4.4.1
Review time	The needed net review time, i.e., the time needed for all the reviews subtracting pauses	ratio	Automatically measured by tool for each review task. Pauses are toggled by the participant.
Review effectiveness	Ratio of defects found by the participant over the total number of defects in the code change (Biffl 2000)	ratio	Computed at the end using the number of detected defects and the total number of defects.
Review efficiency	Number of defects found per hour spent reviewing (Biffl 2000)	ratio	Computed at the end using the number of detected defects and the review time.
Cognitive load	Load imposed on a person's cognitive system while performing a particular task (Paas and Van Merriënboer 1994)	ordinal	see Section 4.2.5

**Table 5** (continued)

Name	Description	Scale	Operationalization
<i>Treated/Measured variables:</i>			
Prof. development experience	Years of experience with professional software development	ordinal	Measured: 6-point scale (“never” ... “11 years or more”), questionnaire
Java experience	Years of experience with the Java programming language	ordinal	Measured: 6-point scale (“never” ... “11 years or more”), questionnaire
Code review experience	The number of years of experience with code reviews	ordinal	Measured: 6-point scale (“never” ... “11 years or more”), questionnaire
Current program. practice	How often participants currently program	ordinal	Measured: 5-point scale (“not” ... “daily or more often”), questionnaire
Current code review practice	How often participants currently perform code reviews	ordinal	Measured: 5-point scale (“not” ... “daily or more often”), questionnaire
Fitness	Perceived tiredness or fitness of the participant before the experiment	ordinal	Measured: 5-point scale (“very tired” ... “very fit”), questionnaire
Experience with jEdit	Whether participants have experience with the jEdit editor	ordinal	Measured: 3-point scale (“none”, “used”, “contributed”), questionnaire
Code change	Code change under review	nominal	Section 4.3.2
Changes order	Whether <i>Large Change A</i> or <i>Large Change B</i> comes first in the experiment.	nominal	Randomized by the tool for each participant.
Usability	Perceived efficiency, effectiveness, and satisfaction in the use of an object (Brooke and et al 1996).	ordinal	Measured: 5-point scale (“agree” ... “disagree”), questionnaire
Understandability	Participants’ understanding of the code. Number of correct answers over all answers per change	ratio	Measured: Multiple choice questions on the reviewed code change.



**Fig. 8** The Experiment Flow

mediator (cognitive load) is “only” measured as in observational studies. Figure 8 shows the flow of our experiment. Differently from what we reported in our registered report (section 3.5) (Gonçalves et al. 2020), we included a short tutorial in the explanation step (step 2) to allow participants to familiarize themselves with the experiment UI. At the start (Step 1 in Fig. 8), developers are briefed on the experiment and the data handling policy.

We ask for informed consent and explicitly request the developers to not share information about the experiment with each other. Then (Step 2), participants are randomly assigned to one of the three treatments and have to complete a short tutorial before proceeding to the review tasks. The tutorial aims to ensure that all participants reach a clear understanding of the tasks, possess a basic level of familiarity with the review platform, and experience their specific guidance approach (if any). A description of the tutorial is given in Section 4.3.1.

After each review, participants are administered a standardized questionnaire (Krell 2017) to measure cognitive load relating to the review they just did (see Section 4.2.5; Step 4 in Fig. 8).

When the participants assigned to the *checklist* or *guided checklist* treatment complete the review tasks, we ask them to answer an adapted version of the System Usability Scale (Brooke and et al 1996) (see Section 4.2.6; Step 5 in Fig. 8).

At the end of the experiment (Step 6), we collect demographic data to gather descriptive characteristics of our sample and intervening variables such as programming and Java experience, coding and reviewing frequency, education, and current stress level.

Developers access the experiment online via a provided URL. Throughout the experiment, we control developers’ comprehension of the system by asking questions about the change under review because code comprehension is an important condition for good reviews (Bacchelli and Bird 2013; Pascarella et al. 2018). The comprehension questions are taken from a previous experiment on code review (Baum et al. 2019) and are described in the related replication package (Baum et al. 2018).

The experiment was conducted in three iterations, with the aim of adjusting the experiment setup, if necessary. A group of ten developers took part in the first iteration of the experiment, while 30 developers were allocated to both the second and third iteration. Apart from asking for participants with higher review experience at the end of the first iteration, we did not make any further adjustments to the experiment.

## 4.6 Analysis

### 4.6.1 Data Cleaning

According to the outline planned in our registered report (Gonçalves et al. 2020), participants who spent less than 5 minutes on a review or did not enter any review remark were classified as NAs (for each review task). We also checked to exclude participants who

restarted the experiment or participated several times (we collect client IPs—hashed to guarantee data anonymization—and cookies). None of the participants was removed as a result of this process. However, three developers did not answer the experiment’s demographics questions. Therefore, we excluded them from the final dataset. This left us with a resulting sample size of 67 developers.

#### 4.6.2 Analysis Plan

In this section, we present our analysis plan (originally stated in our registered report (Gonçalves et al. 2020)). Since the developers in the sample reached low review effectiveness and efficiency and the data provided only limited evidence of the relationships we aimed to investigate, we had to adjust the analysis we could perform significantly, as described in Section 4.7.

In  $RQ_1$ , we perform a One-way ANOVA to identify whether there is a significant difference in code review effectiveness and efficiency among the three treatment groups. Specifics of these differences are explored using Tukey’s Range Test for the post-hoc analysis. In response to  $RQ_1$ , we also present the first regression model used in answering  $RQ_2$  as it refers to the relationship between guidance and performance as well.

We aimed to use mediation analysis (Spencer et al. 2005) for  $RQ_2$  as described and formulated by Imai, Keele and Tingley (Imai et al. 2010). Mediation analysis combines regression models to assess the size of a direct and indirect (mediated) effect of an independent variable on the dependent one. Separate regression models are built for (1) the direct effect of the guidance on code review effectiveness and efficiency, (2) the effect of guidance on cognitive load as a mediator, and (3) the effect of cognitive load on code review performance while controlling for the effect of guidance and other control variables. As last step, (4) a mediation model is built using the regression models as arguments to calculate the significance of the indirect effect.

We planned to construct the models employed in our analysis using the *mediation* R package (Tingley et al. 2014). The type of guidance is considered as the independent variable, code review effectiveness and efficiency as the dependent variables, and the cognitive load as the mediator. To conclude a mediated effect of the treatment on the outcome variable, there must be a significant direct relationship between the treatment and the outcome variables in the model (1) and a significant relationship between the treatment and the mediator and between the mediator and the outcome in models (3) and (4). Model (4) also calculates the overall significance of the path from treatment to the outcome through the mediator (Tingley et al. 2014). If the direct relationship between guidance and code review performance remains significant in the models (3) and (4), we talk about *partial mediation*. If the relationship between the treatment and outcome becomes insignificant due to adding the mediator to the model, we talk about a full mediation (MacKinnon et al. 2007; MacKinnon and Fairchild 2009).

#### 4.6.3 Correlations and Collinearity

Figure 9 presents the statistically significant correlations among core and control variables for our analysis (Pearson correlation,  $p < 0.05$ ). Apart from programming, Java, and code review experience being inter-correlated, we observed a relationship between developers’ understanding of the change (measured as the number of correct answers to questions concerning the reviewed change) and the review time ( $r(48) = .46$ ) as well as a negative

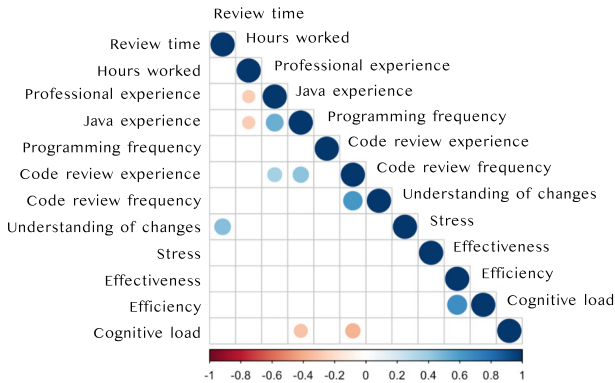


Fig. 9 Significant correlations among independent, dependent and control variables

correlation between cognitive load and Java and code review experience ( $r(48) = -.32$  and  $-.38$ ). In other words, as expected, developers who spent more time on the review had a higher understanding of the change and experienced Java developers and reviewers found the reviews less cognitively demanding. In our sample, developers with more programming and Java experience performed the experiment after fewer hours of work ( $r(48) = -.28$  and  $-.28$ ).

We performed a linear regression to assess which of our control variables are predictors of the output variables, including age, gender, experience with jEdit, and other demographic variables. We found stress, hours worked before the experiment, and review time to be significant predictors of code review effectiveness ( $p < 0.1$ , aggregated for all three changes). More hours worked and lower stress were also predictors of higher code review efficiency ( $p < 0.05$ ) in the *Large Change A* and the hours worked were significant as well in the model aggregating data from all reviews ( $p < 0.1$ ). Furthermore, we calculated the Variance Inflation Factor (VIF), finding no multicollinearity.

### 4.7 Adjustments

The methodology of the study has been pre-registered as a Registered Report at MSR'20 (Gonçalves et al. 2020). We committed to following the pre-approved study design. However, we found out that developers' performance throughout the experiment was deficient despite the proposed treatments. This presented a challenge for the analysis as there was very little variance in the values of code review effectiveness and efficiency. This limited our ability to perform the envisioned mediation analysis. Mediation analysis can be performed only if a relationship between the dependent and independent variable was established. If the relationship is not clearly established, it also cannot be mediated and therefore the mediation analysis becomes unsuitable. This proved to be the case, as reported in Sections 5.1 and 5.2 Mediation analysis is built in several steps, building regression models to investigate the relationships suggested in Fig. 2 and a model where the code review performance is predicted with guidance, cognitive load, and other control variables at the same time (equivalent to model (3) in Section 4.6.2). This sequence of regression models was used to answer the  $RQ_2$ . Furthermore, since one of the regression models investigates the direct relationship between guidance and code review performance (see model (1) in Section 4.6.2), we report its results in answering RQ1.

We attempted to use several strategies to mitigate the low performance of developers. The first attempt was to avoid excluding some participants by trying to pinpoint those who did indeed the task, but just performed poorly. To this aim, we used information about the amount participants scrolled during the experiment as well as their understanding. Participants who were marked as missing values due to lack of review comments or very fast reviewing were included in the analysis with recorded, if they scrolled and answered at least one understanding question after the review correctly. This strategy resulted in several participants being included in the analysis. Several coefficients in our analysis have changed, but it did not improve the quality of the data for analyzing the relationship with code review effectiveness and efficiency as it raised the portion of inefficient developers included in the analysis. Therefore we excluded this approach.

The second strategy we employed was to exclude participants who potentially did not understand the reviewed code enough. We excluded from the analysis developers who did not answer any understanding question in a change correctly. Surprisingly, this strategy resulted in losing several cases of developers who not only entered review comments, but also successfully identified defects and furthered the problem with low variance of values in the variables measuring code review performance.

Seeing that these attempts did not resolve the issues with data quality, we stuck with the original selection criteria and worked with data of developers who spent more than five minutes on the review or entered at least a review remark. In the following sections, we present results based on this sample selection criteria.

All data and materials used in the study are available in our replication package<sup>11</sup>.

## 5 Results

### 5.1 RQ<sub>1</sub>: Does Guidance in Review Lead to a Higher Review Performance?

Our analysis addressed the relationship between guidance and code review effectiveness and efficiency in two ways: (1) comparing the means of the three treatments through a One-Way ANOVA and (2) using a regression model as the first step in the mediation analysis.

The experiment participants showed overall low review effectiveness and efficiency (Table 6, Figs. 10 and 11), which made addressing our research question challenging. To compute developers' review effectiveness and efficiency, we analyzed the aggregated performance in all three review tasks as well as the results of each review change task separately.

The small change had mean effectiveness of 12.5%, while the *Large Change A* ( $M = 7.53\%$ ) and *Large Change B* ( $M = 2.41\%$ ) were more cognitively demanding and developers found fewer defects, as reported in Table 6, Figs. 10, 11, and 12.

Using ANOVA analysis (see Table 7), which compares means of multiple groups, we found the only significant difference to be in the *Large Change B*, where the *checklist* showed significantly better efficiency than the control group ( $p < 0.1$ ) while the guided checklist was not significantly different from neither. The Tukey's Range post-hoc test, also presented in Table 7, did not identify further differences between the three treatments. This was also confirmed in the regression model built to assess the direct relationship between guidance and efficiency, as reported in Table 8. In the *Large Change B*, the use of the

<sup>11</sup>Replication package: <https://doi.org/10.5281/zenodo.5653341>

**Table 6** Descriptives of the main variables by change and treatment

Change	Variable	Total (N=67)		Control (N=29)		Checklist (N=23)		Guided (N=15)	
		Mean	Median	Mean	Median	Mean	Median	Mean	Median
		Small	Effectiveness	12.50	0.00	7.41	0.00	13.64	0.00
	Efficiency	1.01	0.00	0.73	0.00	1.50	0.00	0.80	0.00
	Cognitive Load	4.63	4.50	4.71	4.67	4.60	4.50	4.51	4.50
	Time per review	16.18	13.49	13.75	11.15	13.26	13.39	24.84	17.65
Large A	Effectiveness	7.53	0.00	8.55	0.00	7.41	0.00	5.93	0.00
	Efficiency	0.46	0.00	0.51	0.00	0.61	0.00	0.18	0.00
	Cognitive Load	5.05	5.08	5.14	5.17	4.86	5.00	5.16	5.50
	Time per review	19.50	13.34	14.98	11.52	18.66	11.80	28.35	17.50
Large B	Effectiveness	2.41	0.00	1.20	0.00	4.00	0.00	2.31	0.00
	Efficiency	0.18	0.00	0.08	0.00	0.37	0.00	0.09	0.00
	Cognitive Load	5.03	5.08	5.31	5.58	4.68	4.58	5.04	4.92
	Time per review	19.67	16.58	16.96	13.75	16.75	16.74	29.36	21.86
TOTAL	Effectiveness	7.91	0.00	7.00	3.70	9.70	11.11	7.32	7.04
	Efficiency	0.61	0.29	0.55	0.25	0.96	0.53	0.29	0.22
	Cognitive Load	4.90	4.92	5.05	5.14	4.71	4.97	4.91	4.83
	Time per review	59.48	46.70	50.87	42.14	51.95	46.39	82.07	52.51

Units: Effectiveness - % of found defects; Efficiency - defects found per minute; Cognitive load - Likert scale 0 to 7; Time per review - minutes



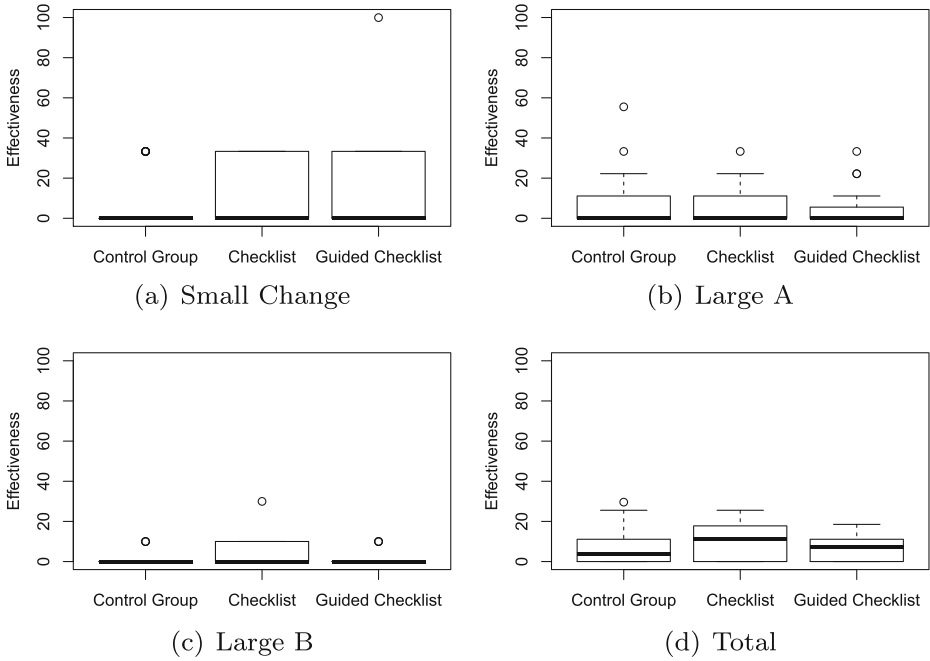


Fig. 10 Effectiveness (in % of found defects) by Guidance Treatment

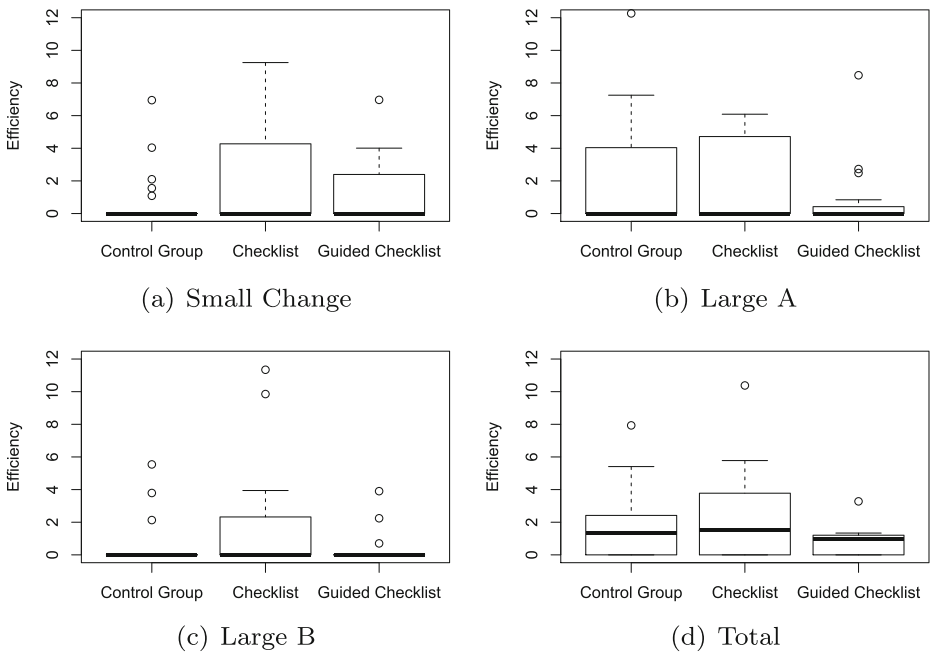
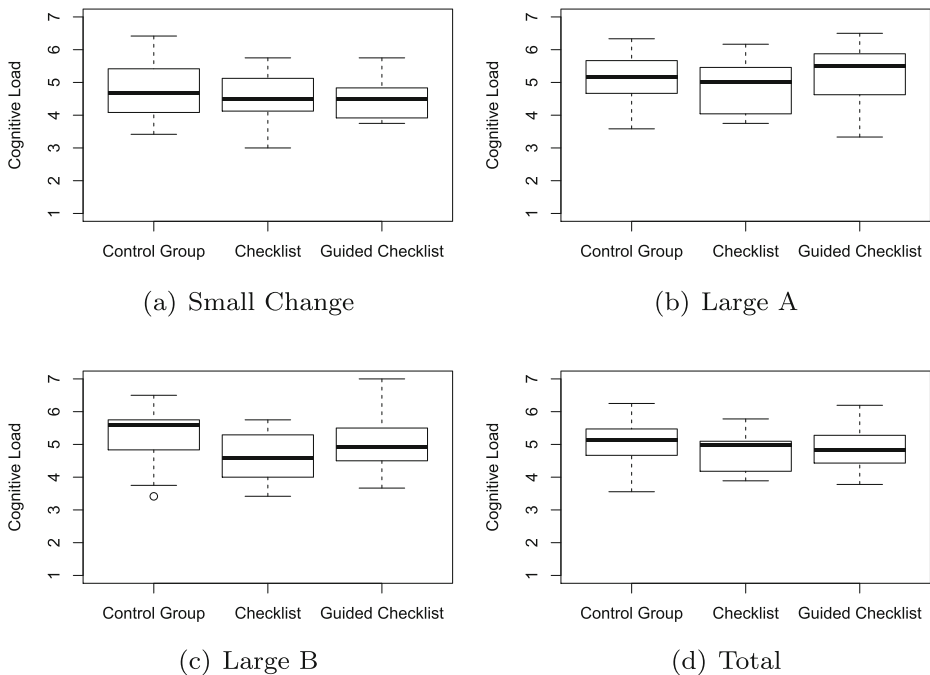


Fig. 11 Efficiency (in defects found per hour) by Guidance Treatment



**Fig. 12** Cognitive Load (on a scale from 1 to 7) by Guidance Treatment

*checklist* is a significant predictor for review effectiveness ( $p < 0.1$ ) and efficiency ( $p < 0.05$ ). It seems that the checklist in the most complex change indeed helped developers to find more defects and to find them in shorter time.

Our regression models also gave us an indication of the presence of a relationship between *guided checklist* usage and code review effectiveness ( $p < 0.05$ ) in the *Small Change*. All the regression models assessing the direct effect of treatments on effectiveness and efficiency are presented in Table 8.

Overall, we cannot conclude that a strong relationship exists between guidance approaches using cognitive load reducing methods and code review effectiveness and efficiency. The developers who participated in our study achieved deficient code review performance, regardless of the treatment to which they were assigned. This situation significantly undermined the possibility of achieving statistically significant results. Nonetheless, our data provide initial indications that the guided checklist effectively supported developers in finding defects in the *small task*, while the checklist allowed them to be more effective and efficient in the more complex review (*Large Change B*), as shown by our regression model. Our ANOVA analysis confirmed that, in the more complex review change-set, the use of the checklist led to better review efficiency compared to the ad-hoc review (control group).

**Answer to RQ1.** The benefits of guidance approaches depend on the complexity of the review change-set. The guided checklist supported developers' effectiveness in the small task, while the checklist increased their effectiveness and efficiency in a more complex review.

**Table 7** ANOVA analysis results and related post-hoc tests - differences between the treatments in effectiveness, efficiency and cognitive load

<i>Effectiveness</i>	ANOVA	Guidance	Residuals	Tukey's Range Test	Check. vs. Control	Guided Check. vs. Control	Guided Check. vs. Check.
	df	2.00	61.00	diff	6.23	12.59	6.36
	F value	2.20		Upper/Lower	-6.80/19.26	-2.02/27.21	-8.83/21.56
Small Change	Sig.			Sig.			
	df	2.00	56.00	diff	-1.14	-2.62	-1.48
	F value	0.23		Upper/Lower	-9.93/7.65	-11.92/6.68	-11.51/8.54
Large Change A	Sig.			Sig.			
	df	2.00	55.00	diff	2.80	1.11	-1.69
	F value	1.53		Upper/Lower	-1.07/6.67	-3.30/5.51	-6.28/2.90
Large Change B	Sig.			Sig.			
	df	2.00	46.00	diff	2.70	0.32	-2.38
	F value	0.46		Upper/Lower	4.48/9.88	-7.18/7.82	-10.43/5.67
Total	Sig.			Sig.			
<i>Efficiency</i>							
	df	2.00	61.00	diff	1.38	0.13	-1.25
	F value	0.88		Upper/Lower	-1.26/4.02	-2.84/3.09	-4.33/1.83
Small Change	Sig.			Sig.			
	df	2.00	56.00	diff	0.54	-1.79	-2.32
	F value	1.18		Upper/Lower	-2.80/3.87	-5.31/1.74	-6.13/1.48
Large Change A	Sig.			Sig.			
	df	2.00	55.00	diff	1.75	0.07	-1.68
	F value	2.54		Upper/Lower	-0.26/3.76	-2.22/2.35	-4.06/0.70
Large Change B	Sig.			Sig.			
	df	2.00	46.00	diff	1.23	-0.99	-2.22
	F value	1.23		Upper/Lower	-1.15/3.62	-3.47/1.50	-4.89/0.45
Total	Sig.			Sig.			

**Table 7** (continued)

Effectiveness	ANOVA	Guidance	Residuals	Tukey's Range Test	Check. vs. Control	Guided Check. vs. Control	Guided Check. vs. Check.
<i>Cognitive Load</i>							
	df	2.00	64.00	diff	-0.11	-0.20	-0.09
	F value	0.38		Upper/Lower	-0.60/0.38	-0.76/0.37	-0.67/0.50
	Sig.			Sig.			
Small Change	df	2.00	64.00	diff	-0.28	0.03	0.31
	F value	1.01		Upper/Lower	-0.81/0.25	-0.58/0.63	-0.32/0.94
	Sig.			Sig.			
Large Change A	df	2.00	64.00	diff	-0.63	-0.26	0.37
	F value	4.17		Upper/Lower	-1.15/ -0.11	-0.85/0.33	-0.25/0.97
	Sig.	*		Sig.	*		
Large Change B	df	2.00	64.00	diff	-0.34	-0.14	0.20
	F value	1.71		Upper/Lower	-0.78/0.10	-0.64/0.36	-0.37/0.72
	Sig.			Sig.			
Total							

Significance codes: '\*\*\*' < 0.001, '\*\*' < 0.01, '\*' < 0.05, '.' < 0.1

**Table 8** Regression results for direct effect of guidance on effectiveness, efficiency and cognitive load

	Effectiveness			Efficiency			Cognitive Load			
	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.	Estimate	S.E.	Sig.	
Small\ Warm-up	Intercept	7.41	3.65	*	1.32	0.74	.	4.71	0.14	***
	Checklist	6.23	5.43		1.38	1.10		-0.11	0.21	
	Guided Checklist	12.59	6.08	*	0.13	1.23		-0.20	0.23	
Large Change A	Intercept	11.76	2.73	***	2.64	1.08	*	5.11	0.18	***
	Checklist	-1.85	3.56		0.56	1.40		-0.28	0.22	
	Guided Checklist	-2.84	3.75		-1.78	1.48		0.03	0.25	
	Order	-6.432	3.033	*	0.24	1.20		0.042	0.195	
Large Change B	Intercept	1.98	1.30		1.15	0.66	.	5.28	0.18	***
	Checklist	2.77	1.60		1.72	0.81	*	-0.62	0.22	**
	Guided Checklist	1.14	1.83		0.09	0.93		-0.26	0.25	
	Order	-1.503	1.405		-1.326	0.713	.	0.04	0.192	
TOTAL	Intercept	9.78	2.23	***	2.11	0.78	**	5.04	0.15	***
	Checklist	2.40	2.85		1.20	0.99		-0.34	0.19	.
	Guided Checklist	0.40	2.97		-0.98	1.03		-0.14	0.21	
	Order	-5.30	2.471	*	-0.51	0.84		0.019	0.162	

Significance codes: \*\*\* $p < 0.001$ , \*\* $p < 0.01$ , \* $p < 0.05$ ,  $p < 0.1$   
 Adjusted R<sup>2</sup>: *Effectiveness*: Small - 0.04, Large A - 0.03, Large B - 0.02, Total - 0.06; *Efficiency*: Small - 0.00, Large A - 0.01, Large B - 0.09, Total - 0.03; *Cognitive Load*: Small - 0.02, Large A - 0.01, Large B - 0.07, Total - 0.01

## 5.2 RQ<sub>2</sub>: Is the Effect of Guidance on Code Review Mediated by a Lower Cognitive Load?

In RQ<sub>2</sub>, we aimed to examine whether the relationship between guidance and code review performance works through lowering the cognitive load. However, the experiment's participants showed overall low review effectiveness and efficiency, and our data did not fulfill the starting condition of an existing relationship between the treatment variable and the outcome (Section 5.1). As we could not perform the mediation analysis, we focused on investigating the individual relationships between guidance and cognitive load, cognitive load and code review performance and effect of guidance and cognitive load on code review performance using regression models.

Examining the direct relationships, we found out that the *checklist* use significantly lowers cognitive load in the most complex change (*Large Change B*,  $p < 0.01$ ) and in the total score for all changes ( $p < 0.1$ ), see Table 8.

Furthermore, using a univariate regression model we established the direct relationship between cognitive load and code review performance, finding that *higher* cognitive load significantly predicts effectiveness in the *Small change* ( $p < 0.05$ ) and *Large Change A* ( $p < 0.1$ ) and also better efficiency in the *Small Change* ( $p < 0.05$ ), as shown in Table 9. These results are surprising and they are further discussed in Section 6.

In the case of *checklist* in the *Large Change B*, we can confirm that the direct effects show that it lowers the cognitive load and improves code review performance. However, the use of the guided checklist showed no statistically significant effect on the level of cognitive load.

After establishing the direct effect of the treatment, we built a regression model to estimate the effect of the treatment on the dependent variable while controlling for the mediator and control variables: *i.e.*, the effect of the guidance, cognitive load, and control variables on

**Table 9** Regression results for the direct effect of cognitive load on effectiveness and efficiency

		Effectiveness			Efficiency		
		Estimate	S.E.	Sig.	Estimate	S.E.	Sig.
Small\Warm-up	Intercept	- 21.75	15.56		- 6.14	1.69	*
	Cognitive Load	7.45	3.35	*	1.73	0.65	*
Large Change A	Intercept	- 7.67	10.00		- 2.35	4.08	
	Cognitive Load	3.57	1.94	.	0.93	0.79	
	Order	- 6.92	2.94	*	0.01	1.20	
Large Change B	Intercept	7.19	5.01		3.93	2.59	
	Cognitive Load	- 0.79	0.97		- 0.42	0.50	
	Order	- 1.51	1.421		- 1.35	0.74	.
TOTAL	Intercept	2.94	10.53		- 1.26	3.77	
	Cognitive Load	1.56	2.09		0.71	0.75	
	Order	- 5.437	2.387	*	- 0.61	0.86	

Significance codes: '\*\*\*' < 0.001, '\*\*' < 0.01, '\*' < 0.05, '.' < 0.1

Adjusted R<sup>2</sup>: *Effectiveness*: Small - 0.06, Large A - 0.09, Large B - 0.00, Total - 0.07; *Efficiency*: Small - 0.09, Large A - 0.01, Large B - 0.04, Total - 0.01

code review effectiveness and efficiency. As reported in Section 4.6.3, the control variables for effectiveness are stress, review time in minutes, and hours worked before the experiment. Stress and hours worked are intervening variables for efficiency as well. The resulting models are presented in Tables 10 and 11.

When including the control variables, some of the direct effects disappeared while others emerged. In the *Small Change*, higher cognitive load remains related to higher effectiveness ( $p < 0.1$ ) and efficiency ( $p < 0.01$ ). However, the direct effect of the *guided checklist* on review effectiveness cannot be observed anymore. The use of the *guided checklist* was significantly related to a *lower* review effectiveness in the *Large Change A* ( $p < 0.1$ ), suggesting that the guided checklist was not helpful for this change. *Checklist* remained a significant predictor of code review efficiency in *Large Change B* ( $p < 0.05$ ). The *checklist* use and higher cognitive load were also predictors of efficiency considering the aggregated performance of all three review tasks.

The control variables predict code review performance too. A lower level of stress and a higher amount of hours worked before taking part in the experiment are related to higher code review effectiveness (*Large Change A*; all three reviews). More hours worked are also related to higher review efficiency (*Large Change A and B*). Furthermore, developers who spent more time on the review of *Large Change A* were more effective.

In our RQ2, despite not being able to perform a mediation analysis because of the low review performance of the participants, we still collected insights about the relationship between guidance, cognitive load, and code review performance. We (1) observed that higher cognitive load is a statistically significant predictor of review effectiveness (in the *Small Change* and *Large Change A*) and review efficiency (*Small Change*), and (2) provided initial evidence for the mediation effect for checklists as they improve review performance while reducing developers' cognitive load.

**Answer to RQ2.** The participants' low review performance prevented us from performing a mediation analysis. The devised regression models showed that (1) higher cognitive load is associated with higher performance in the small change and higher effectiveness in a complex change; (2) the checklist seems to improve review performance and reduce developers' cognitive load.

### 5.3 Checklists Usability

The *checklist* and *guided checklist* achieved similar usability scores ( $M = 57.35$  and  $M = 58.5$ ). Moreover, the scale assigns a letter grade to the assessed system. Both of our treatments were rated as *D* (Poor usability). Even though the resulting score is not optimal, the lower performance of the control group developers seems to indicate that a poor implementation of these treatments is not the main reason for the low performance. The ratings of the individual items are reported in Fig. 13.

Developers reported that they would need more time to familiarize themselves with the *checklist*, while the *guided checklist* was reported as easier to learn to use. Moreover, the *guided checklist* was more positively evaluated regarding its integration into the experiment UI. Both guidance approaches (checklist and guided checklist) were reported as easy to use and developers felt confident in their use.

**Table 10** Regression results for the effect of guidance and cognitive load on effectiveness while controlling for other variables

		Effectiveness		
		Estimate	S.E.	Sig.
Small\Warm-up	Intercept	-24.60	15.97	
	Checklist	7.26	5.30	
	Guided Checklist	4.32	6.14	
	Cognitive Load	6.63	3.37	.
	Stress	-4.31	2.76	
	Review Time	0.06	0.07	
	Hours worked	0.88	0.93	
Large Change A	Intercept	-6.39	11.39	
	Checklist	-1.06	3.24	
	Guided Checklist	-6.41	3.59	.
	Cognitive Load	2.31	1.87	
	Stress	-3.99	1.67	*
	Review Time	0.21	0.09	*
	Hours worked	1.05	0.55	.
	Order	-0.012	2.99	
Large Change B	Intercept	-0.47	6.67	
	Checklist	2.86	1.79	
	Guided Checklist	0.00	2.05	
	Cognitive Load	-0.09	1.07	
	Stress	-0.20	0.86	
	Review Time	0.07	0.06	
	Hours worked	0.33	0.30	
	Order	-2.37	1.54	
TOTAL	Intercept	-4.12	11.07	
	Checklist	3.57	2.66	
	Guided Checklist	-2.13	2.98	
	Cognitive Load	1.70	2.00	
	Stress	-2.99	1.30	*
	Review Time	1.00	0.44	
	Hours worked	1.00	0.44	*
	Order	-5.63	2.18	*

Significance codes: '\*\*\*\*' < 0.001, '\*\*\*' < 0.01, '\*\*' < 0.05, '.' < 0.1

Adjusted R<sup>2</sup>: Small- 0.14, Large A - 0.23, Large B - 0.00, Total - 0.25

We also observed significant correlations between usability and other variables in the analysis. Among checklist users, developers who code more frequently reported a better usability of the checklist ( $r(22) = 0.5, p < 0.05$ ). Also developers with higher cognitive load found the *checklist* more usable ( $r(22) = 0.35, p < 0.05$ ).



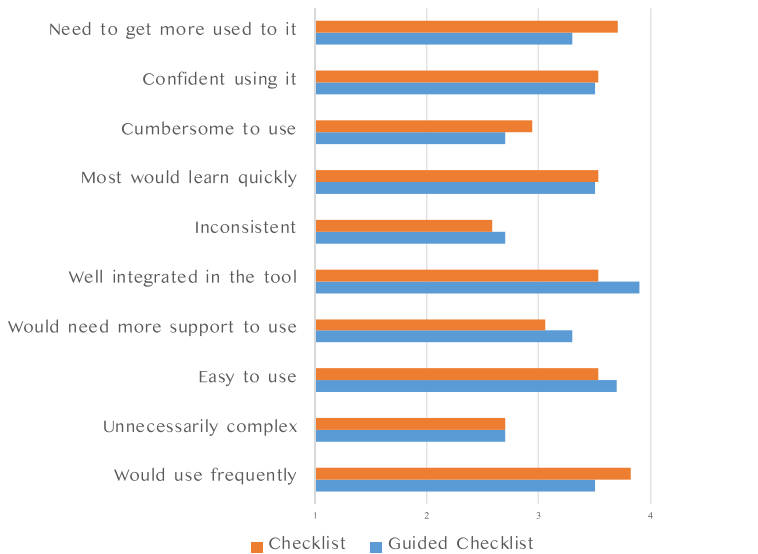
**Table 11** Regression results for the effect of guidance and cognitive load on efficiency while controlling for other variables

		Efficiency		
		Estimate	S.E.	Sig.
Small\Warm-up	Intercept	-7.30	3.36	*
	Checklist	1.58	1.06	
	Guided Checklist	0.27	1.19	
	Cognitive Load	1.84	0.67	**
	Stress	-0.64	0.59	
	Hours worked	0.13	0.19	
Large Change A	Intercept	-2.61	4.69	
	Checklist	1.07	1.33	
	Guided Checklist	-2.09	1.39	
	Cognitive Load	0.96	0.75	
	Stress	-1.72	0.69	*
	Hours worked	0.40	0.22	.
	Order	-0.056	1.121	
Large Change B	Intercept	-1.01	3.33	
	Checklist	1.85	0.90	*
	Guided Checklist	-0.06	0.97	
	Cognitive Load	0.04	0.54	
	Stress	0.08	0.43	
	Hours worked	0.25	0.15	.
	Order	-1.506	0.723	*
TOTAL	Intercept	-3.29	4.01	
	Checklist	1.72	0.97	.
	Guided Checklist	-1.16	1.01	
	Cognitive Load	0.84	0.72	
	Stress	-0.88	0.47	.
	Hours worked	0.35	0.16	*
	Order	-0.772	0.783	

Significance codes: '\*\*\*\*' < 0.001, '\*\*\*' < 0.01, '\*\*' < 0.05, '.' < 0.1

Adjusted R<sup>2</sup>: Small - 0.09, Large A - 0.12, Large B - 0.09, Total - 0.16

Developers assigned to the *guided checklist* took, on average, half an hour longer to complete the experiment compared to the developers in the control group or the *checklist* group. Users of the *guided checklist* reported lower usability with increasing time spent on the reviews ( $r(15) = -0.45$ ,  $p < 0.05$ ). Therefore, the time overhead posed by the *guided checklist* compared to the other two treatments decreased its usability. The longer time spent on the reviews potentially put an extra strain on the need of developers to hold the reviewed code in their working memory and increased the representational holding demands. We did not observe relationships between usability and code review performance.



**Fig. 13** Mean scores of individual System Usability Scale items

## 6 Discussion and Lessons Learned

The developers who took part in our study achieved an overall low review performance (both in terms of review effectiveness and efficiency) regardless of the treatment to which they were assigned. Nevertheless, we observed significant relationships in the experiment data that allowed us to draw initial conclusions (1) on the benefits of providing guidance to developers during code review to lower cognitive load and (2) on the role of cognitive load in reaching code review effectiveness and efficiency. Furthermore, we collected valuable lessons learned to conduct future studies on this topic.

**Review guidance reduces cognitive load:** Checklists and software development strategies aim to improve developers' performance by lowering the cognitive load (Kamsties and Lott 1995; LaToza et al. 2020). However, this relationship has not been explicitly tested yet. The results of our experiment provided initial results on how guidance can lower the cognitive load of developers. Based on the literature, we expected the reduction of the cognitive load to play a fundamental role in preventing cognitive overload in more complex changes (Banerter 2002). We indeed observed that the checklist significantly lowers cognitive load in the most complex review change (*Large Change B*) and on the whole review (considering all review tasks together). Our results gave an initial indication that guidance indeed lowers reviewers' cognitive load, indicating that further research could be valuable to be conducted in this research direction. The review strategy did not prove to lower the cognitive load for novice reviewers. Nonetheless, further studies need to be conducted to collect insight on the relationship between guidance and cognitive load.

**Higher cognitive load can improve performance.** Even though checklist usage lowers cognitive load, higher cognitive load predicted both code review effectiveness and efficiency in the *Small Change*. This stands in opposition to the hypothesis based on the literature suggesting that lower cognitive load leads to improved code review performance, as

suggested by previous work in the field (Baum et al. 2017b; Pascarella et al. 2018; Kamsties and Lott 1995). The subject changes showed different patterns in the results: While in the *Small Change*, the guided checklist and higher cognitive load were more helpful, in the *large changes*, the checklist proved to be more efficient and to lower cognitive load, while the guided checklist led to lower effectiveness. This seems to indicate that the change complexity plays an important role in which type of guidance developers require.

*Lesson learned 1. Developers may need a different type of assistance depending on the complexity of a change-set.*

If confirmed in further studies, this finding can have consequences for both researchers and practitioners: The complexity of the review change-set under analysis is a significant factor for choosing the right guidance approach to support the review.

According to the literature, lowering the cognitive load is important to prevent the upper-limit scenario when working memory is overloaded, therefore saving up limited cognitive resources for effective and efficient performance (Bannert 2002). However, we have observed a scenario where developers did not perform well regardless of the treatment they were assigned to. Furthermore, a higher cognitive load was linked to better performance: We found that a higher cognitive load led to higher review effectiveness and efficiency in the *Small Change*. We interpret this as the need to get engaged in the task and invest the cognitive resources into actually being able to identify the defects. If this finding is confirmed in further studies, when devising guidance approaches, researchers should take into account the positive effect that cognitive load might have on review performance. Future research should investigate how to best balance cognitive load to improve developers' effectiveness and efficiency.

*Lesson learned 2. Cognitive load should not overload the working memory capacity, however it may be needed to perform well in a task.*

**Code understanding is fundamental:** Previous studies reported how understanding the code is indeed one of the main challenges that developers face during code review (Tao et al. 2012; Bacchelli and Bird 2013). For this reason, researchers devised numerous approaches to increase reviewers' understanding: e.g., re-ordering review changes (Baum et al. 2019; 2017b) or untangling complex review change-sets (Barnett et al. 2015; Dias et al. 2015). The former approach aims at increasing the understandability of a review change-set by showing changes in a more meaningful order (as opposed to the alphabetical one currently offered by popular code review tools: e.g., Gerrit or Phabricator). The latter focuses instead on dividing large review change-sets into smaller ones, comprising only changes related to the same issue.

We noticed that developers in this experiment faced significant issues in answering correctly the understanding questions shown at the end of each review task. As reported in Table 2, in the *Small Change* participants achieved an average score of 1.21 out of 2 when answering the understanding questions at the end of the task. In the *Large Change A* and *Large Change B*, developers achieved an average of 1.45 (out of 4) and 1.37 (out of 3) correct answers. These results seem to indicate that despite the support provided by review guidance, a significant increase in review performance can not be achieved if reviewers struggle to understand the content of a review change-set. It seems reasonable to think that

for review guidance approaches to be effective, it is necessary to ensure that developers possess a good understanding of the code.

This finding might have practical implications on how to support developers during code review: Researchers must not only focus on guiding developers during the review but also support reviewers in gaining a preliminary understanding on the content of a review change-set.

*Lesson learned 3. A sound understanding of the review change-set is necessary even if guidance approaches are provided.*

**The experiment tasks must fit the abilities of participants:** The user interface and code reviews have been previously successfully implemented in an experiment with a sample of developers collected online and in a company (Baum et al. 2019). However, the developers in our sample not only performed poorly, but also had problems to answer the questions about understanding the code (as reported in Table 2). Furthermore, we have found a significant correlation between lower cognitive load and programming and reviewing experience (as shown in Fig. 9). Therefore, there is a not negligible possibility that the reviews were too difficult for these developers. This might explain why higher cognitive load was actually a significant predictor of code review performance. The developers actually needed to put a considerable mental effort into comprehending and successfully reviewing the code. To be more successful and also provide more diverse data, a future experiment should test more types of changes and defects with the target developers before selecting the appropriate changes for the final experiment.

*Lesson learned 4. The difficulty of the review tasks needs to match the ability of study participants either by fitting the task difficulty to the available participants or using a sample of developers with sufficient skills.*

**Autonomy adds value to guidance:** Our results indicate that the guided checklist performed better in the simpler task and was unhelpful in the *Large Change A*, while the simple checklist was more effective in the complex task. As reported in Section 5.1, the devised regression models highlighted the existence of a relationship between the guided checklist and review effectiveness in the *Small Change*, while no effect was reported for the two complex changes. At the same time, the use of the checklist was shown to be correlated with higher review performance (effectiveness and efficiency) in *Large Change B*. Given that the guided checklist implemented additional methods to reduce cognitive load compared to the checklist, these results are surprising. We believe the reason for this difference lies in developers' *autonomy*.

While the checklist allows developers to have maximum flexibility on how to check the items, the devised guided checklist controlled the flow of review explicitly telling participants what to check and when. This makes the guided checklist useful for a shorter detailed review, but it might become overwhelming for longer and more complex reviews. The importance of autonomy seems to be confirmed by LaToza et al. (LaToza et al. 2020), who designed a tool to support explicit strategies to perform software development tasks. In particular, their solution supported autonomous execution of these strategies and did not enforce the steps, rather allowed the developers to define the steps and be flexible in navigating the code.

The importance of autonomy should be taken into account by both researchers and practitioners (e.g., project managers) when implementing approaches to improve the code review process of a project. Approaches that too strictly guide developers may deplete their cognitive resources by enforcing a too fine-grained level of review, not allowing reviewers to adapt the review style to their personal needs. This might undermine the support this kind of approaches aim to offer.

*Lesson learned 5. Future studies can be designed to compare the effects of autonomy over a rigid control of the review flow in more complex tasks.*

**Familiarity with guidance takes time:** The results of our System Usability Scale questionnaire (reported in Fig. 13) show that participants believed our guidance approaches to be well-integrated in the online experiment platform (the corresponding item achieved a mean score of 3.53 for the checklist and 3.9 for the guided checklist). Nonetheless, developers reported difficulties and the need for more support in learning how to use them. Participants assigned a mean score of 3.7 (for the checklist) and 3.3 (for the guided checklist) to the “need to get more used to it” item in the SUS questionnaire. This indicates that, despite the linearity of the implemented guidance approaches and the presence of a tutorial on how to use them, participants still would have benefited from more time to familiarize with these tools. Future studies could take into account this factor and either plan for longer controlled experiments or use different kinds of studies (e.g., field studies). For example, a longitudinal study would give developers time to familiarize themselves with the guidance approach under investigation.

*Lesson learned 6. In a single controlled experiment, developers might not have enough time to familiarize themselves with the proposed guidance approach. Longer controlled experiments or field studies can be a suitable alternative to mitigate this issue.*

## 6.1 Actual vs. Expected results

**RQ1.** Our first aim was to investigate the effect of guidance approaches (checklist and guided checklist) on developers’ review performance. We expected these approaches to lead to higher review effectiveness and efficiency on all three review tasks. We hypothesized to observe a lower increase in performance on the small review task compared to the two more complex review tasks: The importance of lowering the cognitive load is greater when developers perform more complex tasks and potentially might reach the cognitive overload.

However, our results showed that the guidance approaches do not lead to better participants’ performance on all three review changes. Our checklist increased developers’ review effectiveness and efficiency only on *Large change B*, while the guided checklist led to higher effectiveness only on the small change. Furthermore, our findings highlighted the importance of developers’ *autonomy* (see Lesson learned 5) while performing code review. A more guided approach (guided checklist) was proven effective on a small review change-set but unhelpful on a larger change-set, potentially because of the extra time overload required by the detailed level of review and lower usability related to it, as reported in Section 5.3.

**RQ2.** In RQ2, we expected our mediation analysis to show that review guidance (checklist and guided checklist) leads to better performance by lowering developers' cognitive load. Even though both guidance approaches led to better performance (albeit in different contexts), only the checklist lowered developers' cognitive load (in the *Large change B* and in the total score). Only in the *Large change B*, the most complex change, we could observe the mediation pattern hypothesized in the literature – using the checklist lowered the cognitive load and also led to better performance.

Moreover, we expected that lower cognitive load always leads to better review performance. However, our results showed the opposite: A higher cognitive load was related to better review effectiveness and efficiency in the small change, and to better effectiveness in *Large Change A*. These results seem to indicate that cognitive resources also need to be invested to perform good reviews.

## 6.2 Influence of Participants' Lack of Experience

In this section, we reflect on the possible effects that the lack of experience in code review among our participants may have had on our results. In our experiment, we recorded both *programming experience* and *review experience* of the participants. While our participants had professional programming experience with Java, they can be considered as *novice reviewers*. Therefore, it seems that programming experience does not directly translate into the ability to find defects during code review. For this reason, in the following reflection, we focus on code review experience.

Experience reduces the cognitive load of the reviewers (Section 2): With higher experience, fewer cognitive resources are needed to complete a review. Developers with no review experience lack the automatization on what and how to review, how to find appropriate information, and how to process it. Therefore, their cognitive resources are depleted faster. This might have an impact on the type of guidance inexperienced developers need as opposed to experienced reviewers. A more guided review approach, as the one offered by our guided checklist, might be beneficial for novice reviewers as it guides them in the review step by step: The guided checklist supported developers in achieving higher review effectiveness in the *Small Change*. However, it did not assist the novice reviewers well in the complex change. This result is potentially due to the fact that the guided checklist requires a very detailed and thorough process that leads to a longer review time. Therefore, there is an additional strain for the mental load caused by representational holding – the need to keep information ready in the working memory for a prolonged period of time.

Developers' experience with the system under review might also have had an impact on our results. In our experiment, participants had to review change-sets extracted from a system they were *not* familiar with. This might have led them to spend significant cognitive resources on understanding the review changes, increasing their cognitive load and undermining the effect of the guidance. Therefore, future work can be designed and executed to investigate whether the devised guidance approaches achieve better results when applied to review change-sets with whom developers are already familiar.

## 7 Threats to Validity

**Construct validity:** The set of review tasks might influence developers' results. To mitigate this issue, we employed code review tasks already successfully applied in a previous

experiment on code review (Baum et al. 2019). Moreover, participants had to use an online platform and guidance treatment with which they were not familiar before. This might have negatively affected their review performance. To reduce this bias, the online platform used showed review changes in a similar fashion to the one of popular code review tools (e.g., Gerrit or Phabricator). Moreover, before starting the experiment, participants had to complete a short tutorial explaining the use of the experiment UI and of the guidance approaches (checklist and guided checklist).

The way in which the checklist and guided checklist were implemented might have introduced bias in our results. We developed our guidance approaches following best practices from both researchers (Chernak 1996; Degani and Wiener 1991; Kamsties and Lott 1995) and industry (McConnell 2004). Nonetheless, we can not rule out that different guidance approaches (e.g., with a more specific focus on developers' autonomy) could lead to different results.

Our guided checklist made developers spend significantly longer time than the other groups. Furthermore, with a longer time, users of the guided checklist reported lower usability. Therefore, the review time was entered as an important confounding factor in the regression model.

**Internal validity:** We analyzed the experiment logs to identify participants who did not take the experiment seriously. To this aim, we disregarded participants who spent less than 5 minutes doing the review or did not enter any remark. Moreover, we also controlled for developers who might have taken part in the experiment several times or have restarted the experiment.

A poor understanding of the use of the experiment UI and the code under review might have introduced bias in our results. To mitigate this issue, we supported participants in two ways. (1) We showed them an interactive tutorial on the UI and used guidance approaches (if they were assigned to one of them). Participants were required to complete the tutorial by interacting with the UI, before being able to proceed with the experiment. Furthermore, we asked them questions about the instructions of the experiment. If they answered wrongly, we displayed the correct answer. (2) Prior to each review task, we showed participants a description of the context of the change. Moreover, we asked questions to the participants to verify their correct understanding of the context of the change. As done before, if developers answered these questions wrongly, we made them aware of the correct answer. Furthermore, we controlled developers' understanding of the code through a set of questions at the end of each review task.

Participants in our experiment achieved overall low review effectiveness and efficiency, regardless of the treatment to which they were assigned (control, checklist, and guided checklist). This prevented us from drawing strong conclusions to answer our research questions. Nonetheless, we were able to collect indications on the benefits of review guidance over developers' review performance and cognitive load.

**External validity:** All participants in the experiment were professional developers with experience in Java. However, they had rather low experience with code reviews and code review effectiveness and efficiency. Therefore, our results are bound to novice reviewers with limited ability to identify defects.

Furthermore, all participants work in the same company and, to the best of our knowledge, possess a very similar technical and cultural background. This might limit the generalizability of our findings.

The majority of the participants ( $N = 39$ ) worked at least eight hours before taking part in our experiment. This might have negatively influenced their review performance

and, therefore, introduced a bias in our results. To check the possible effect of the number of hours worked on the review performance, we included this variable as a control variable in our regression models. Our results showed that more hours worked do not have a negative effect on the reviewers' performance. On the contrary, they led to better performance.

We hypothesize three reasons as to why more hours worked before the experiment can be related to better code review performance: (1) The participants of the experiment are more productive at the end of their working day, (2) more productive developers worked on the experiment later in the day, and (3) in our sample, younger developers with more frequent coding practice had more hours worked before they started the experiment (Section 4.6.3); therefore, developers with more coding practice were working on the experiment later in the day.

## 8 Conclusion

We have examined how two types of guidance incorporating different methods of lowering cognitive load relate to code review performance and cognitive load. While a checklist performed better in a complex task, a tool-supported strategic checklist execution proved to be more effective in the simple task. Moreover, we obtained an initial indication that the use of a checklist lowers developers' cognitive load. However, a higher cognitive load was related to better code review performance. The study participants achieved low code review effectiveness and efficiency as well as a limited understanding of the code. Therefore, the higher cognitive load was probably needed to achieve better performance. Further studies are still needed to investigate the relationship between guidance, cognitive load, and code review performance.

## Appendix

### A.1 Checklist items

**Code Review Checklist** Please use the following checklist for reviewing the presented change. For each question check:

- “Yes” field ? in case you have checked the code and everything is OK
- “No” field ? in case you have checked the code and found some defects

If the item of the checklist does not apply or you did not check the code for it, please leave both columns unchecked.

#### General

Yes	No	
<input type="checkbox"/>	<input type="checkbox"/>	Are all the changes relevant?
<input type="checkbox"/>	<input type="checkbox"/>	Do the classes and methods fulfill their purpose?
<input type="checkbox"/>	<input type="checkbox"/>	Are the messages and texts for the user correct?



**Classes**

- | <i>Yes</i>               | <i>No</i>                |   |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | Are all assignments of attributes correct?  |
| <input type="checkbox"/> | <input type="checkbox"/> | Are the classes implemented correctly?<br>E.g., do they correctly implement inheritance and are they protected<br>against unintentional access to their data? |

**Methods**

- | <i>Yes</i>               | <i>No</i>                |   |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | Do methods always return a valid value?               |
| <input type="checkbox"/> | <input type="checkbox"/> | Do methods check parameters for validity (if needed)? |
| <input type="checkbox"/> | <input type="checkbox"/> | Are all parameters used?                              |

**Arguments**

- | <i>Yes</i>               | <i>No</i>                |   |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | Are the correct arguments used in all method calls? |

**Variables**

- | <i>Yes</i>               | <i>No</i>                |   |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | Are all variables, counters and accumulators initialized properly and, if<br>necessary, re-initialized each time they are used? |
| <input type="checkbox"/> | <input type="checkbox"/> | Are all declared variables being used?  |

**If-Then statements**

- | <i>Yes</i>               | <i>No</i>                |   |
|--------------------------|--------------------------|---|
| <input type="checkbox"/> | <input type="checkbox"/> | Do the if-else statements fit the intended purpose? |
| <input type="checkbox"/> | <input type="checkbox"/> | Are all edge cases handled?                         |

**Loops**

- | <i>Yes</i>               | <i>No</i>                |  |
|--------------------------|--------------------------|--|
| <input type="checkbox"/> | <input type="checkbox"/> | Do the loops end under all possible conditions?      |
| <input type="checkbox"/> | <input type="checkbox"/> | Are the break and continue statements used properly? |

**Recursion**

- | <i>Yes</i>               | <i>No</i>                |  |
|--------------------------|--------------------------|--|
| <input type="checkbox"/> | <input type="checkbox"/> | Does the recursion terminate properly? |

**Errors**

- | <i>Yes</i>               | <i>No</i>                |                                       |
|--------------------------|--------------------------|---------------------------------------|
| <input type="checkbox"/> | <input type="checkbox"/> | Are the exceptions handled correctly? |

## Final Check

Yes      No  
            Are all the changes consistent with each other?

**Funding** Open access funding provided by University of Zurich. P. Wurzel Gonçalves, E. Fregnan, and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2\_170529.

**Availability of data and material** The methodology of the present study was accepted as registered report at MSR 2020. The published article is available here: <https://dl.acm.org/doi/abs/10.1145/3379597.3387509>

All data and materials are available in our replication package at the following link: <https://doi.org/10.5281/zenodo.5653341>

**Code Availability** The code developed in the context of this study is available in our replication package at the following link: <https://figshare.com/s/b26d1936417fe2c2c257>

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Abdelnabi Z, Cantone G, Ciolkowski M, Rombach D (2004) Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate. In: Proceedings 2004 international symposium on empirical software engineering, 2004. ISESE'04. IEEE, pp 239–248
- Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 international conference on software engineering. IEEE Press, pp 712–721
- Bannert M (2002) Managing cognitive load—recent trends in cognitive load theory. *Learning and Instruction* 12(1):139–146
- Barnett M, Bird C, Brunet J, Lahiri SK (2015) Helping developers help themselves: Automatic decomposition of code review changesets. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1. IEEE, pp 134–144
- Basili V, Caldiera G, Lanubile F, Shull F (1996) Studies on reading techniques. In: Proceedings of the twenty-first annual software engineering workshop, vol 96, p 002
- Baum T (2019) Cognitive-support code review tools: improved efficiency of change-based code review by guiding and assisting reviewers. PhD thesis, Hannover: Institutionelles Repositorium der Universität Hannover. <https://doi.org/10.15488/9164>
- Baum T, Schneider K (2016) On the need for a new generation of code review tools. In: International conference on product-focused software process improvement. Springer, pp 301–308
- Baum T, Leßmann H, Schneider K (2017a) The choice of code review process: A survey on the state of the practice. In: Product-focused software process improvement. Springer International Publishing, Cham, pp 111–127
- Baum T, Schneider K, Bacchelli A (2017b) On the optimal order of reading source code changes for review. In: 2017 IEEE International conference on software maintenance and evolution (ICSME). IEEE, pp 329–340
- Baum T, Schneider K, Bacchelli A (2018) Online material for Associating working memory capacity and code change ordering with code review performance. <https://doi.org/10.6084/m9.figshare.5808609>

- Baum T, Schneider K, Bacchelli A (2019) Associating working memory capacity and code change ordering with code review performance. *Empir Softw Eng* 24(4):1762–1798
- Biffi S (2000) Analysis of the impact of reading technique and inspector capability on individual inspection performance. In: Software engineering conference, 2000. APSEC. 2000. Proceedings. Seventh Asia-Pacific. IEEE, pp 136–145
- Brooke J et al (1996) Sus—a quick and dirty usability scale. *Usability evaluation in industry* 189(194):4–7
- van Bruggen JM, Kirschner PA, Jochems W (2002) External representation of argumentation in cscl and the management of cognitive load. *Learning and Instruction* 12(1):121–138
- Carver JC (2003) The impact of educational background and experience on software inspections. PhD thesis, University of Maryland
- Chernak Y (1996) A statistical approach to the inspection checklist formal synthesis and improvement. *IEEE Trans Softw Eng* 22(12):866–874
- Ciurumelea A, Proksch S, Gall HC (2020) Suggesting comment completions for python using neural language models. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 456–467
- Cohen J (1992) Statistical power analysis. *Current Directions in Psychological Science* 1(3):98–101
- Cohen J (2010) Modern code review Oram A, Wilson G (eds), vol 18, Making Software, O'Reilly, chap
- Degani A, Wiener EL (1991) Human factors of flight-deck checklists: the normal checklist. Tech. rep
- DeLeeuw KE, Mayer RE (2008) A comparison of three measures of cognitive load: Evidence for separable measures of intrinsic, extraneous, and germane load. *Journal of educational psychology* 100(1):223
- Denger C, Ciolkowski M, Lanubile F (2004) Does active guidance improve software inspections? a preliminary empirical study. In: IASTED Conf. on software engineering, pp 408–413
- Dias M, Bacchelli A, Gousios G, Cassou D, Ducasse S (2015) Untangling fine-grained code changes. In: 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER). IEEE, pp 341–350
- Dobbs AR, Rule BG (1989) Adult age differences in working memory. *Psychology and aging* 4(4):500
- Dunsmore A, Roper M, Wood M (2001) Systematic object-oriented inspection—an empirical study. In: Proceedings of the 23rd international conference on software engineering. IEEE Computer Society, pp 135–144
- Dunsmore A, Roper M, Wood M (2003) Practical code inspection techniques for object-oriented systems: an experimental comparison. *IEEE Software* 20(4):21–29
- Ebad SA (2017) Inspection reading techniques applied to software artifacts—a systematic review. *Comput Syst Sci Eng* 32(3):213–226
- Fagan M (2002) Design and code inspections to reduce errors in program development. In: Software pioneers. Springer, pp 575–607
- Gonçalves PW, Fregnan E, Baum T, Schneider K, Bacchelli A (2020) Do explicit review strategies improve code review performance? In: Proceedings of the 17th international conference on mining software repositories, pp 606–610
- Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp 345–355
- Greiler M (2021) How code reviews work at microsoft. <https://www.michaelagreiler.com/code-reviews-at-microsoft-how-to-code-review-at-a-large-software-company/>
- Gridnev A (2017) Examples of code review checklists and guides. <https://andriegridnev.medium.com/examples-of-code-review-checklists-and-guides-2dfed082a86d>
- Gutha SR (2015) Code review checklist – to perform effective code reviews. <https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/>
- Imai K, Keele L, Tingley D (2010) A general approach to causal mediation analysis. *Psychological Methods* 15(4):309
- Jedlitschka A, Ciolkowski M, Pfahl D (2008) Reporting experiments in software engineering. In: Guide to advanced empirical software engineering. Springer, pp 201–228
- Kamsties E, Lott CM (1995) An empirical evaluation of three defect-detection techniques. In: European software engineering conference. Springer, pp 362–383
- Khandelwal S, Sripada SK, Reddy YR (2017) Impact of gamification on code review process: An experimental study. In: Proceedings of the 10th innovations in software engineering conference, pp 122–126
- Ko AJ, LaToza TD, Hull S, Ko EA, Kwok W, Quichocho J, Akkaraju H, Pandit R (2019) Teaching explicit programming strategies to adolescents. In: Proceedings of the 50th ACM technical symposium on computer science education, pp 469–475
- Krell M (2017) Evaluating an instrument to measure mental load and mental effort considering different sources of validity evidence. *Cogent Education* 4(1):1280256

- Kvålseth TO (1989) Note on cohen's kappa. *Psychological Reports* 65(1):223–226
- Lanubile F, Mallardo T, Calefato F, Denger C, Ciolkowski M (2004) Assessing the impact of active guidance for defect detection: a replicated experiment. In: 10Th international symposium on software metrics 2004 Proceedings. IEEE, pp 269–278
- LaToza TD, Arab M, Loksa D, Ko AJ (2020) Explicit programming strategies. *Empir Softw Eng* 25(4):2416–2449
- MacKinnon DP, Fairchild AJ (2009) Current directions in mediation analysis. *Current Directions in Psychological Science* 18(1):16–20
- MacKinnon DP, Fairchild AJ, Fritz MS (2007) Mediation analysis. *Annu Rev Psychol* 58:593–614
- MacLeod L, Greiler M, Storey MA, Bird C, Czerwonka J (2017) Code reviewing in the trenches: Challenges and best practices. *IEEE Softw* 35(4):34–42
- Matthews G, Wohleber R, Lin J (2019) Stress, skilled performance, and expertise: Overload and beyond. *The Oxford handbook of expertise*, 1–39
- Mayer RE, Moreno R (2003) Nine ways to reduce cognitive load in multimedia learning. *Educational Psychologist* 38(1):43–52
- McCabe TJ (1976) A complexity measure. *IEEE Trans Softw Eng* 4:308–320
- McConnell S (2004) *Code complete*. Pearson Education, London
- McMeekin DA, von KonskyBR, Robey M, Cooper DJ (2009) The significance of participant experience when evaluating software inspection techniques. In: 2009 Australian software engineering conference. IEEE, pp 200–209
- Oosterwaal S, Deursen Av, Coelho R, Sawant AA, Bacchelli A (2016) Visualizing code and coverage changes for code review. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 1038–1041
- Paas F, Tuovinen JE, Tabbers H, Van Gerven PW (2003) Cognitive load measurement as a means to advance cognitive load theory. *Educational Psychologist* 38(1):63–71
- Paas FG, Van Merriënboer JJ (1994) Instructional control of cognitive load in the training of complex cognitive tasks. *Educational Psychology Review* 6(4):351–371
- Pascarella L, Spadini D, Palomba F, Bruntink M, Bacchelli A (2018) Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2(CSCW):1–27
- Rafi DM, Moses KKR, Petersen K, Mäntylä MV (2012) Benefits And limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7Th international workshop on automation of software test (AST). IEEE, pp 36–42
- Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, pp 202–212
- Rong G, Li J, Xie M, Zheng T (2012) The effect of checklist in code review for inexperienced students: An empirical study. In: 2012 IEEE 25th conference on software engineering education and training. IEEE, pp 120–124
- Rothlisberger D, Harry M, Binder W, Moret P, Ansaloni D, Villazon A, Nierstrasz O (2012) Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *IEEE Trans Softw Eng* 38(3):579–591
- Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: a case study at google. In: Proceedings of the 40th international conference on software engineering: Software engineering in practice, pp 181–190
- Spencer SJ, Zanna MP, Fong GT (2005) Establishing a causal chain: why experiments are often more effective than mediational analyses in examining psychological processes. *Journal of Personality and Social Psychology* 89(6):845
- Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th working conference on mining software repositories. IEEE, pp 180–190
- Tao Y, Dang Y, Xie T, Zhang D, Kim S (2012) How do software engineers understand code changes? an exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, pp 1–11
- Thelin T, Runeson P, Wohlin C (2003) An experimental comparison of usage-based and checklist-based reading. *IEEE Trans Softw Eng* 29(8):687–704
- Tingley D, Yamamoto T, Hirose K, Keele L, Imai K (2014) Mediation: R package for causal mediation analysis. *Journal of Statistical Software*
- Tymchuk Y, Mocchi A, Lanza M (2015) Code Review: veni, vidi, vici. In: 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER). IEEE, pp 151–160
- Unkelos-Shpigel N, Hadar I (2015) Gamifying software engineering tasks based on cognitive principles: The case of code review. In: 2015 IEEE/ACM 8th international workshop on cooperative and human aspects of software engineering. IEEE, pp 119–120

- UsabiliTEST (2020) System usability scale (sus) plus. <https://www.usabilitytest.com/system-usability-scale>
- Uwano H, Nakamura M, Monden A, Matsumoto K (2006) Analyzing individual performance of source code review using reviewers' eye movement. In: Proceedings of the 2006 symposium on Eye tracking research & applications, pp 133–140
- Zhang T, Song M, Pinedo J, Kim M (2015) Interactive code review for systematic changes. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1. IEEE, pp 111–122

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Pavlina Wurzel Gonçalves** is a PhD student at the Institute of Informatics at the University of Zurich, Switzerland. She received a Master's degree in Psychology from the Faculty of Social Science, Masaryk University, Brno, Czech Republic in 2017. Her research work focuses on leveraging human factors to improve code review process through finding ways to help developers use their cognitive resources efficiently, and communicate and collaborate in an effective way.



**Enrico Fregnan** is a Ph.D. student in the Zurich Empirical Software engineering Team (ZEST) at the University of Zurich. He received his bachelor's degree at Politecnico di Milano, Italy and his master's degree at Delft University of Technology, The Netherlands. His research focuses on investigating how to support developers during code review.



**Tobias Baum** works as director at SET GmbH, a software company in Hannover, Germany. He received his PhD from Leibniz Universität Hannover and a M. Eng. in computer science from FHDW Hannover. His research interests include better cognitive support for change-based code review, further improvements of code review processes, and more generally improving the efficiency of software engineering in small and medium-sized enterprises.




**Kurt Schneider** is a full professor of software engineering at Leibniz Universität Hannover, Germany. He received his Doctoral degree from the University of Stuttgart, Germany. He held a postdoctoral position at the Center for LifeLong Learning and Design (L3D) at the University of Colorado at Boulder, USA. From 1996 to 2003, he was a researcher and manager at the Daimler research center in Ulm, Germany. His research interests include requirements engineering, software quality and the role of human interaction and communication in software engineering.



**Alberto Bacchelli** received the bachelor's and master's degrees in computer science from the University of Bologna, Italy, and the PhD degree in software engineering from the Università della Svizzera Italiana, Switzerland. He is an associate professor of Empirical Software Engineering with the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland.

## Affiliations

Pavlina Wurzel Gonçalves<sup>1</sup> · Enrico Fregnan<sup>1</sup>  · Tobias Baum<sup>2</sup> · Kurt Schneider<sup>2</sup> · Alberto Bacchelli<sup>1</sup>

Enrico Fregnan  
fregnan@ifi.uzh.ch

Tobias Baum  
tobias.baum@inf.uni-hannover.de

Kurt Schneider  
kurt.schneider@inf.uni-hannover.de

Alberto Bacchelli  
bacchelli@ifi.uzh.ch

<sup>1</sup> University of Zurich, Zurich, Switzerland

<sup>2</sup> Leibniz Universität Hannover, Hannover, Germany