The Journal of Systems and Software 241 (2026) 113004

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

The prevalence of code review guidelines for GUI-based testing in open-source☆

Andreas Bauer [a,c,*], Florian Angermeier [a,b], Emil Alégroth [a], Seth Anglert [d]

[a] Software Engineering Research Lab (SERL), Blekinge Institute of Technology, Sweden
[b] fortiss, Germany
[c] Technische Hochschule Nürnberg Georg Simon Ohm, Germany
[d] Independent Researcher, Sweden

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.17104676

Keywords:
GUI-based testing
GUI testing
Test scripts
Code review
Sample study
Mining software repositories
GitHub

ABSTRACT

Context: Code review has become a core practice in collaborative software engineering, helping ensure code quality, detecting potential bugs, and supporting communication among developers. Prior research has shown that code review practices differ between production and test code, suggesting that established code review guidelines may fall short in the context of test and GUI-based test code. Particularly, GUI-based testing lacks adequate support during the code review process. To address this, we proposed a set of code review guidelines specifically designed for reviewing GUI-based test files. However, these guidelines have not yet been empirically evaluated, which limits their practical relevance.
Objective: This study examines the extent to which code review comments on GUI-based tests align (explicitly or implicitly) with the concerns captured by the proposed guidelines and uses the findings to refine these guidelines.
Method: To achieve this, we sampled code review comments discussing GUI-based test files across 100 open-source projects and manually analyzed 1000 pull requests to determine the extent to which the reviewers' comments align with the proposed guidelines.
Results: Review comments aligned with the proposed guidelines in 808 of 1000 pull requests. We found empirical support for 25 of the 33 guidelines in code review practice. The most frequently observed guideline concerns the correct use of testing techniques and exception handling, particularly regarding locators, explicit waits, and timeout behavior.
Conclusion: The observed alignment suggests that the proposed guidelines capture concerns articulated in practice, indicating practical relevance for GUI-based test reviews. The results position the guidelines as practice-informed recommendations and motivate future work to evaluate their impact on review quality and outcomes.

1. Introduction

Modern software systems tend to be large and complex, evolving rapidly to the point where a single developer cannot oversee all aspects of the software or fully understand the implications of every change. Thus, the development of modern software systems requires extensive collaboration among professionals with diverse technical skills and domain knowledge to manage the increasing complexity of developed systems (Randel and Jaussi, 2003; Layzell et al., 2000; Na et al., 2004). To facilitate collaborative efforts, many industry and open-source projects have embraced code review as informal and asynchronous discussions about changes and their potential impacts before they are merged into the codebase (Davila and Nunes, 2021; Bacchelli and Bird, 2013; McIntosh et al., 2016; Barnett et al., 2015; Gousios et al., 2014). In addition to improving the quality of artifacts under review, code reviews facilitate knowledge sharing among developers and testers.

Software testing requires contributions from both technical and non-technical experts (Sánchez-Gordón and Colomo-Palacios, 2018). Automated software tests are developed, reviewed, and maintained alongside production code, necessitating collaborative efforts similar to those applied to the production code. If tests are not reviewed to identify subpar tests and human errors, the quality of tests declines, and

☆ Editor: Yan Cai.

* Corresponding author at: Software Engineering Research Lab (SERL), Blekinge Institute of Technology, Sweden.
E-mail addresses: andreas.bauer@bth.se (A. Bauer), angermeir@fortiss.org (F. Angermeir), emil.alegroth@bth.se (E. Alégroth).

https://doi.org/10.1016/j.jss.2026.113004
Received 23 November 2025; Received in revised form 30 April 2026; Accepted 8 June 2026
Available online 4 July 2026
0164-1212/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

maintenance costs increase (Berner et al., 2005; Garousi and Felderer, 2016; Kruchten et al., 2012). Among the various testing approaches, graphical user interface (GUI)-based testing stands out as a technique for verifying and validating a system's behavior through its visual interface, simulating real-world user interactions (Alégroth and Feldt, 2017; Coppola and Alégroth, 2022).

Existing research on code review primarily focuses on production code and low-level test code as the main artifacts being reviewed (Davila and Nunes, 2021; Spadini et al., 2019, 2018). Spadini et al. (2018) identified differences in the review processes for production code and test code. When reviewing test code, reviewers tend to focus on test-specific aspects, such as improving testing practices, identifying tested and untested paths, and evaluating assertions within the test code. Additionally, the lack of test-specific information in code review tools often forces developers to examine tests locally within their integrated development environment (IDE), where they can utilize navigation and other features that are not available in code review tools.

Conducting a code review is a complex task that requires reviewers to assess not only the quality of the specific artifact under review but also its broader implications for the entire system (Tao et al., 2012). Additionally, code review is a cognitively demanding activity. A high cognitive load can negatively impact performance on the task (Matthews et al., 2019), ultimately affecting the quality of the review.

Best practices and guidelines for code review are available in both open-source and industrial communities (Dong et al., 2021), indicating their importance to these groups. However, the survey conducted by Dong et al. (2021) does not include information on best practices specifically related to tests in general or GUI-based tests in particular.

To address the gap in missing code review guidelines for GUI-based tests, we proposed 33 guidelines specific to the code review of GUI-based tests in our previous exploratory work (Bauer et al., 2023). These guidelines were derived from a systematic literature review encompassing academic white literature as well as gray literature, with the goal of analyzing all relevant sources related to code review guidelines. This provided us with a systematic account of the available guidelines in the scientific community for different artifact types, including production code, test code, and GUI-based test code. One finding from the literature review was the lack of specific guidelines for GUI-based tests. This motivated us to map guidelines for production and test code to GUI-based tests. However, the guidelines have not yet been empirically evaluated, which means we cannot recommend them with confidence to practitioners.

Building on our previous work (Bauer et al., 2023), this study examines whether and how the concerns embodied in the proposed guidelines are reflected in real-world code review discussions. We analyze the extent to which code review comments on pull requests (PRs) that modify GUI-based tests align with the previously proposed guidelines. We define alignment as either explicit, when a comment directly reflects a guideline's concern, or implicit, when a comment could be addressed by a guideline. Observing such alignment would suggest that the guidelines capture concerns relevant to practitioners and inform refinements of the guidelines.

This study thus represents the first empirical step toward evaluating the previously proposed guidelines. We propose that guidelines for code review in GUI-based test files can offer benefits similar to those of general guidelines for production code. We believe that tailored guidelines for GUI-based tests could enhance the effectiveness of the review process and the quality of the reviewed artifacts. For testers, they clarify expectations and explicate the information needs of reviewers. For reviewers, they serve as a checklist for systematically evaluating test artifacts and identifying potential quality concerns.

We claim the following contributions of this study:

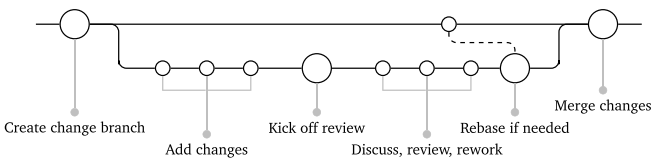

Fig. 1. Development workflow of integrating changes with code reviews.

1. An empirical assessment of the alignment between proposed guidelines for reviewing GUI-based test files and code review comments from real-world projects.
2. Concrete examples that illustrate each guideline as observed in review comments, and resulting refinements to the guidelines.
3. A comparative analysis of our findings alongside those reported in other studies.
4. A replication package containing all scripts used for data collection and a set of intermediate artifacts, allowing the replication and extension of our study by other researchers.

2. Background and related work

2.1. Code review

Code review, as it is practiced nowadays, is characterized as informal, tool-supported, and lightweight processes (Cohen, 2010). In contrast, earlier methods of code review, often referred to as code inspections, followed a more formal, waterfall-like process (Fagan, 1976). This approach was associated with overhead costs, which hindered its adoption.

Code reviews have been shown to be effective at raising software quality and promoting knowledge sharing (Fatima et al., 2020; McIntosh et al., 2016; Baker, 1997). Defects found and resolved during code reviews often enhance the software's understandability and modifiability, rather than its visible functionality (Mantyla and Lassenius, 2009). In addition to technical benefits, code reviews provide socio-technical advantages. They help justify changes, keep the team informed about the project's progress, and serve as a record of design decisions. The practice of code review is prevalent in open-source software (Pascarella et al., 2018) and is also adopted by leading software companies such as Microsoft (Bacchelli and Bird, 2013) and Google (Sadowski et al., 2018).

Code reviews are not tied to any specific tool and can even be conducted manually, though they are most often facilitated by dedicated tools, such as Gerrit.[1] Modern code hosting services, such as GitHub[2] and GitLab,[3] integrate code review directly into their core collaborative workflows, typically through a pull or merge request mechanism.

In contemporary development practices, changes to the codebase are usually made in separate branches. Once the changes are ready to be merged into the main branch, a review is initiated and assigned to one or more reviewers. Reviewers examine the proposed modifications, provide feedback, and may request revisions or additional context to fully understand the purpose and impact of the changes. This process often involves multiple iterations until the reviewers are satisfied. At that point, the changes are either approved and merged into the main branch or rejected if they do not meet the required standards. Fig. 1 illustrates such a workflow.

[1] https://www.gerritcodereview.com.
[2] https://github.com.
[3] https://about.gitlab.com.

2.2. GUI-based testing

GUI-based testing is a form of testing performed at a higher level of abstraction, in which a system's behavior is verified through its graphical user interface (GUI), mirroring user interactions (Coppola and Alégroth, 2022; Alégroth and Feldt, 2017). By interacting with the GUI, GUI-based testing enables the assessment of both the visual appearance and functional correctness of the system under test (SUT) (Alégroth and Feldt, 2017). In industry, such testing is performed manually as well as through automated tests (Garousi and Mäntylä, 2016; Garousi et al., 2020; Itkonen et al., 2009; Engström et al., 2010; Haas et al., 2021; Leitner et al., 2007).

GUI-based testing can be categorized in several ways (Alégroth et al., 2015). One common approach categorizes tests based on how the test cases are defined, generally falling into three categories: scripted tests, capture-and-replay, and model-based tests. Scripted GUI-based testing involves the development of test scripts or code using specialized automation APIs, tools, and frameworks, such as Selenium, Playwright, and Cypress. Capture-and-replay testing relies on tools that record the actions performed by a tester or user on the GUI of the system. These tools create re-executable test sequences from the recorded operations. Model-based testing involves the generation of models that represent the SUT's intended behavior from a GUI perspective. These models are used to generate test sequences that ensure coverage of various states of the GUI. The models created can be visual (defined with nodes and vertices), textual (as used in behavior-driven development), or formal (for instance, in mathematical modeling).

This study specifically focuses on automated GUI-based testing, where test cases are written as scripts along with supporting artifacts that enable automation. Other forms of test automation, such as model-based GUI testing, where tests are defined using visual or textual models, are excluded from this study. This focus is motivated by the fact that code-based test scripts closely resemble production code, which is the typical subject of code review practices.

Additionally, GUI-based testing can be applied to the GUI of various types of applications, including web applications, native desktop applications (Savic et al., 2022), and mobile applications (Coppola et al., 2020, 2019). Web applications are a common focus for GUI-based testing, and a wide range of testing tools is available for this purpose. Therefore, this study concentrates exclusively on web applications and exclude other types. However, our findings could be relevant, to some extent, to both mobile and desktop applications that are built on top of web technologies.

A GUI test case is composed of two components: (a) the initial GUI state in which the test case is executed, and (b) a series of events that act as the test input for the GUI (Memon, 2001).

2.3. Related work

In our previous study, we conducted a multivocal literature review of both academic and gray literature to identify code review guidelines for GUI-based testing files (Bauer et al., 2023). Software engineering guidelines outline best practices for producing high-quality software. However, most existing code review practices and guidelines aim to enhance the effectiveness and efficiency of code reviews and focus primarily on source code and lower-level test code (Dong et al., 2021; Bauer et al., 2023).

We discovered that there were no specific code review guidelines for GUI-based test files, revealing a gap in the academic literature. As a result, we adapted guidelines from production and low-level test code reviews to apply them, where possible and relevant, to GUI-based testing. This process resulted in the creation of 33 guidelines, organized into nine categories. Table 1 provides an overview of the nine guideline categories, including the number of guidelines in each category and their support in both white and gray literature.

Table 1
Categories of code review guidelines for GUI-based tests, along with the number of guidelines (#G) and the supporting sources in each category (#WL = white literature, #GL = gray literature) (Bauer et al., 2023).

ID	Guideline category	#G	#WL	#GL
G1	Perform automated checks	2	27	32
G2	Use checklists	1	8	19
G3	Provide context information	9	44	54
G4	Utilize metrics	3	11	11
G5	Ensure readability	7	14	51
G6	Visualize changes	3	14	3
G7	Reduce complexity	3	34	32
G8	Check conformity with the requirements	1	0	3
G9	Follow design principles and patterns	4	0	18

The most relevant work related to this topic is by Spadini et al. (2018). In this study, the authors investigated whether and how developers review test files during code reviews. Through interviews with 12 experienced practitioners, the study revealed that reviewing test files differs from reviewing production code. Specifically, when reviewing test files, reviewers tend to focus on test-specific concerns, such as adherence to testing best practices, coverage of expected and edge-case behaviors, identification of tested and untested paths, and the clarity and correctness of assertions. While the study highlights the difference between production and test code review, the study does not distinguish among different types of tests, i.e., unit, integration, or system tests. Therefore, it remains unclear whether review practices differ even more based on the type of test. For example, unit tests typically emphasize correctness and isolation of individual components, while system tests focus on the end-to-end behavior of a system and integration across modules. The cognitive load required to review a system's behavior may be greater than that of isolated components, potentially requiring tailored review guidelines or tooling support. Given that cognition plays an important role in software engineering activities (Fagerholm et al., 2022; Gonçales et al., 2021, 2019), research is investigating its impact on tasks such as interpreting different levels of abstraction (Minas et al., 2017).

In a study conducted by Gonçalves et al. (2022) (based on the registered report Gonçalves et al., 2020), the authors investigate the impact of a guidance approach via a checklist on the effectiveness and efficiency of code reviews through experimentation. The research suggests that providing a guidance approach for code reviews is expected to reduce the cognitive load associated with performing these reviews. The experimental setup included three types of conditions: ad hoc review, a checklist, and a guided checklist.

For effectiveness, the researchers measured the number of functional defects identified, while for efficiency, they assessed the number of defects found in relation to the review time taken. The checklist comprised 17 binary checks (indicating either that everything is okay or that defects have been found) and is organized into topics such as general, classes, methods, arguments, variables, if-then statements, loops, recursion, errors, and a final check. The guided checklist presented the same items as the regular checklist, but did not show all items simultaneously. Instead, it displayed only those relevant to the selected code segment.

The study found no strong correlation between the guidance provided and code review performance. While the checklist has the potential to reduce developers' cognitive load, it was observed that a higher cognitive load might have led to better performance, possibly due to the overall low effectiveness and efficiency of the study participants.

In contrast to the previously proposed guidelines, their checklist was designed for general production code reviews. The review of test code and GUI-based test code occurs less frequently (Spadini et al., 2018; Bauer et al., 2025), which limits opportunities to gain expertise through reviews. Thus, we assume that the potential benefits of guidelines supporting code reviews may be greater.

2.4. Previous evaluation attempts

In a previous attempt to evaluate the proposed guidelines, we conducted a pilot study to determine whether code review guidelines specific to GUI-based testing improve both the quantity and quality of review comments. This pilot study was a quasi-experiment involving four industry testing experts. The experimental setup, however, did not yield meaningful insights into how the guidelines affected the code review process. This was partly due to the small sample size of testing experts, but more importantly, because the guidelines derived from the literature were too general and needed refinement. For instance, what specific feedback do reviewers offer to ensure the proper use of testing techniques and exception handling? This study aims to outline the necessary steps and provide real-world examples and refined guidelines for future experimental studies.

3. Methodology

We conduct a theory-informed qualitative content analysis of code review comments to assess whether the concerns captured by our previously proposed guidelines are reflected in real-world review practice. Our goal is not to validate the guidelines' effectiveness or quantify their causal impact on the code review process. Rather, we examine their empirical grounding by investigating whether and how review comments on GUI-based test files align with the proposed guidelines.

This approach guides our research question:

RQ: To what extent, and in what ways, do code review comments on real-world projects align with the proposed guidelines for reviewing GUI-based test files (explicitly or implicitly)?

An observational design cannot establish that applying these guidelines improves code review outcomes. However, it can determine whether the guidelines capture concerns that practitioners actually raise. If concerns represented by the guidelines recur across many independent repositories, projects, and reviewers, this suggests that the guidelines reflect practice-relevant review knowledge. We therefore interpret recurring alignment as evidence of practical relevance and empirical grounding, not as proof of effectiveness.

According to the ABC framework proposed by Stol and Fitzgerald (2018), this study is classified as a sample study, as we aim to draw conclusions applicable to a larger population of open-source software projects. Sample studies are commonly employed to analyze extensive collections of software development artifacts or projects, especially within the realm of open-source software.

A preliminary analysis[4] of three open-source projects revealed that a small number of pull requests (≤1%) discuss GUI-based test files during the code reviews. In 35 instances, code review comments mention GUI-based test files that can be mapped to the previously proposed guidelines. Most of the code review comments we found focused on understandability and readability. Other comments addressed the robustness and maintenance of test cases, the need for proper validation, and requests for additional contextual information.

Building on these initial insights, we expand the scope of our investigation to include multiple repositories, providing a broader, more nuanced picture of code review comments related to GUI-based testing, including cases where such comments are absent, in real-world code reviews. This analysis also guides further qualitative exploration and potential refinement of the previously proposed guidelines.

[4] The analysis was conducted to determine if findings from interviews could be complemented by findings from open-source repositories in our previous study (Bauer et al., 2025).

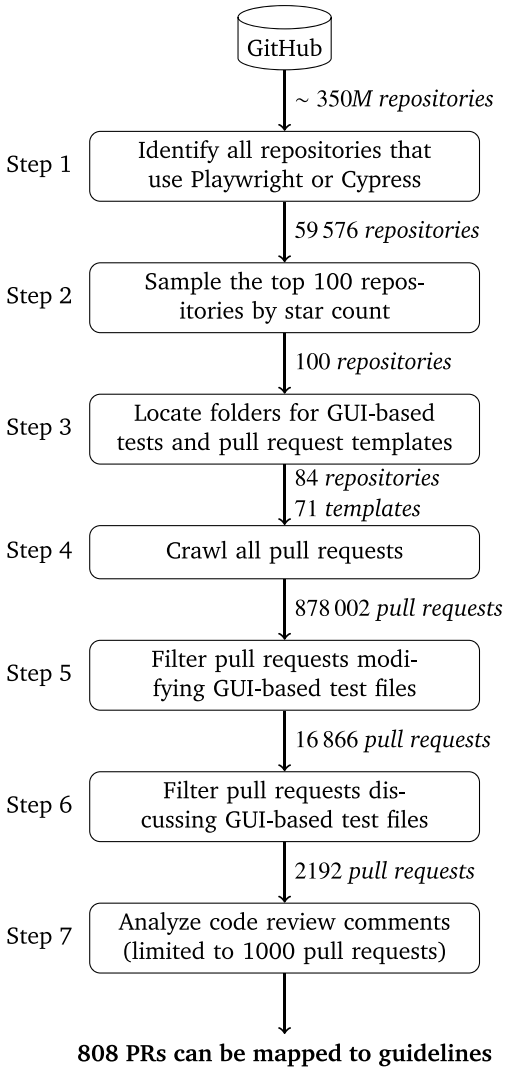

Fig. 2. Summary GitHub review analysis.

3.1. Mining Software Repositories (MSR)

To collect relevant evidence for the proposed GUI-based testing guidelines, we employ software repository mining as a research method (Hassan, 2006), analyzing code review comments from open-source GitHub projects.

In this section, we outline all the steps of the data-gathering process. Fig. 2 illustrates a visual summary of this process. For transparency and replicability, all scripts used to perform the data gathering are available in the study's replication package (Bauer and Angermeir, 2026).

Inclusion of testing tools. Before we begin identifying open-source repositories, we clarify the selection of Cypress and Playwright as testing tools to search for repositories that include them. In academic studies, Selenium (Gojare et al., 2015) is frequently referenced as a browser automation tool to facilitate automated testing, such as GUI-based testing (Niranjanamurthy et al., 2018; García et al., 2024). Selenium, a general-purpose browser automation library, poses several challenges for end-to-end testing, including issues of flakiness (Leotta et al., 2023). More recent tools, such as Cypress, Playwright, and Puppeteer, have emerged to address some of the challenges associated with Selenium and have surpassed it in terms of usage and popularity (García et al., 2024).

Table 2
Search query to find repositories that include configuration files for Playwright or Cypress. This query locates files whose names include the specified `filename` and whose extensions are `.js` or `.ts`. Example of files that will be found: `playwright.config.ts`, `playwright.config.js`, and `example.cypress.config.ts`.

API	`GET api.github.com/search/code?q=Query`
Query	`filename:playwright.config or filename:cypress.config extension:ts OR extension:js`

This popularity trend is also reflected in the State of JS 2024 developer survey (Greif, 2024), which collected responses from 14 015 participants regarding the use of JavaScript tools/frameworks for front-end web applications. In the testing tool category, Cypress ranked 3rd, Playwright ranked 7th, and Selenium ranked 9th out of thirteen based on usage. While Cypress and Playwright have seen increased adoption in recent years, Selenium has experienced a steady decline. Other tools in the testing tool category lack native browser automation and instead rely on or incorporate Playwright and Cypress for end-to-end GUI-based testing in a browser environment. For instance, Jest (ranked 1st) uses Puppeteer, Storybook (ranked 2nd) uses Cypress/Playwright, and Vitest (ranked 4th) uses Playwright.

We selected Cypress and Playwright based on their popularity and their ability to perform GUI-based testing without integrating other testing tools. This focus is intended to enhance the relevance of our study for future industry and open-source project practice. Additionally, the findings may generalize to the other tools that utilize Cypress and Playwright.

Step 1: Identify repositories. We begin by searching for repositories that include files related to testing frameworks designed for GUI-based testing. In particular, our focus is on configuration files for Playwright and Cypress, as these files have standardized names that can serve as a reliable marker for the search, unlike the test case names or associated metadata. To facilitate future studies, such as extensions or replications, the scripts can be configured to search for configuration files from other tools. To conduct the search, we utilized GitHub's REST API[5] to identify repositories that contain the desired configuration files. The specific search query we used is detailed in Table 2. Since the GitHub API limits each request to 1000 results (repositories), we employed a pagination strategy based on file size. By specifying file-size ranges for the targeted configuration files, we partitioned the results into manageable subsets of repositories that are below the maximum search limit. This approach enabled a sliding-window mechanism, where each file-size range represents a window, to systematically retrieve all matching repositories. As a result of this step, we identified a total of 59 576 repositories.

Step 2: Sample top 100 repositories. The resulting repository list from Step 1 includes GitHub's star counts, which act as a proxy for popularity. This star count allows us to sort and select the most popular repositories. Popular repositories tend to be widely used and we therefore assume that they are more likely to be well tested than less popular repositories. Since not all steps of the analysis process can be automated, and because crawling data through GitHub's API is time-intensive, we need to limit the subsequent analysis to a manageable number of repositories. Thus, we ranked repositories by star count and selected the top 100.

Step 3: Locate GUI-based test directories and pull request templates. The first author manually identified directories containing GUI-based test files. These directory paths serve as filters for relevant pull requests in subsequent analysis. To make it easier to look up test files later, we also noted the test file extension (e.g., `*.test.ts`, `*.spec.ts`) for each repository to account for cases where test and production code are organized in the same directory. Repositories that use Playwright or Cypress solely for non-GUI tests were excluded from further analysis. This includes scenarios where tests are executed against a web API or where the integration of single components in an isolated context is assessed. Out of the 100 repositories, 84 remain for further analysis. We considered replacing the excluded repositories with additional repositories to maintain a sample size of 100. However, as we extracted and filtered the pull requests from the 84 repositories in the subsequent steps, we realized we had already collected more relevant pull requests than we could analyze manually. Therefore, any extra effort spent on adding more repositories would not have provided meaningful benefits.

We also analyzed the template files used for pull requests. When a repository includes pull request templates, contributors automatically see the template's content in the pull request body. They are expected to provide the requested information, e.g., the rationale of the change. On GitHub, pull request templates must be named `pull_request_template.md`. We collected all templates from the top 100 repositories and organized them into a single folder for easy inspection and comparison. Afterward, we manually reviewed each template and counted how often specific topics appeared in them.

Step 4: Crawl all pull requests. In this step, we collected all pull request data from 84 of the top 100 repositories.[6] We utilized GitHub's GraphQL API,[7] instead of using the REST API, to gather comprehensive data for each pull request, as it enabled us to customize the information included in the responses. This allowed us to extract all relevant information for the study, including the title, description, the comment thread with references to modified files, individual comments, and a list of changed files.

Initially, we retrieved only the first 100 changed files per pull request to optimize simultaneous retrieval of multiple pull requests. Later, we revisited these pull requests to ensure that we had not overlooked any relevant data. Ultimately, we re-gathered data from GitHub and analyzed 14 381 pull requests that affected more than 100 files. This process resulted in the addition of 108 relevant pull requests to the final dataset.

Step 5: Filter pull requests modifying GUI-based test files. For each repository, we excluded pull requests that did not create, modify, or delete GUI-based test files. This filtering was automated and relied on the directory paths identified in Step 3 and the list of changed files for each pull request.

Step 6: Filter pull requests discussing GUI-based test files. Not all pull requests that modify GUI-based test files will include a code review of those files. Therefore, we need to filter the pull requests that actually discuss GUI-based test files via code review comments.

On GitHub, there are two ways to provide comments: threads and general comments. Thread comments consist of one comment or a series of comments related to a specific file being modified, making it easy to identify what is under discussion. In contrast, general comments do not reference any specific file and typically address the entire pull request.

To inform our filtering criteria, the first author manually reviewed over 500 randomly sampled PRs. From this preliminary analysis, we found that general comments rarely contained substantive discussion of GUI-based test files, instead focusing on merge conflicts or vague mentions of test flakiness without technical justification that could be mapped to one of the proposed guidelines. As a result, we limited inclusion to PRs containing thread comments on GUI-based test files.

[5] https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28.

[6] We encountered many issues collecting pull request data from the `elastic/kibana` repository as the API responded with an error when resolving the changed files of a pull request. Thus, we substituted it with the repository ranked 101.

[7] https://docs.github.com/en/graphql.

While this excludes potentially relevant general comments, it enables a manual, high-fidelity analysis without introducing the complexities and potential inaccuracies associated with (semi-)automated comment classification. Additionally, we excluded automated bot comments, such as those reporting test metrics or flagging contributor license agreement (CLA) requirements, as well as comments that consist solely of emojis, as they do not represent human review.

The resulting set of 2192 relevant PRs was compiled into a spreadsheet and shared among the authors for further analysis (Step 7).

Step 7: Aligning code review comments with guidelines. We mapped comments on GUI-based test files to the 33 guidelines organized into nine categories from our prior study (Bauer et al., 2023). For mapping, we used the existing explanations for each guideline, including its purpose, description, and examples of its application, to assess whether a comment aligns with a specific guideline.

A comment was considered aligned if it directly or indirectly reflected a guideline. For example, if a comment includes a link to the Playwright documentation, it aligns with guideline G3.5, which states that links to relevant resources and documentation should be provided. Additionally, a comment may also be aligned if the issue raised can be addressed by a guideline. For example, if a comment requests a better description of a test case to convey its intent more clearly, it could be improved by following guideline G5.4, which emphasizes adhering to coding style and naming practices. We mapped comments to specific guidelines (e.g., G3.5) rather than to broader categories (e.g., G3). In the Appendix, we provide illustrative examples of how code review comments were mapped to guidelines and the underlying decision rationale.

The first two authors collaborated on the mapping process over multiple sessions, discussing each case to minimize overlooked content and resolve any ambiguities. When a comment was unclear, we examined the associated pull request in GitHub to gain a comprehensive understanding of the changed tests. In cases where we examined the entire PR on GitHub, our exclusion of general comments did not appear to have a negative impact, as we rarely found general comments that could be mapped to a specific guideline. Decisions regarding the mapping of specific comments were documented to ensure consistent mapping throughout all sessions. Our goal during the mapping was to preserve the intent and nuances of code review comments. To achieve this, we adopted a collaborative process in which differing author interpretations were discussed and resolved immediately, preventing inconsistencies in the mapping. Naturally, this collaborative process posed the risk of introducing biases. To address this bias, we assessed reproducibility with an independent coder afterward.

We limited the analysis to 1000 pull requests due to the substantial manual effort required. To analyze these 1000 pull requests, we spent close to 21 person-hours: seven sessions of approximately 90 min each with two people. In this final sample, we ensured that all repositories containing comments about GUI-based test files were represented. The mapping of each pull request to the guidelines is made transparent in the replication package. It includes a spreadsheet listing all pull requests related to GUI-based test files, along with guideline mappings, additional notes, and URLs to access the pull requests on GitHub.

Reproducibility with independent coder. To assess the reproducibility of our coding, we invited an independent coder and evaluated inter-rater agreement and reliability. Based on the coding in Step 7, we operationalized and documented the applied coding schema as a codebook that outlines when to apply each code, when not to use it, and provides examples of how each code could appear in the data.

The new coder, who is neither an author of this manuscript nor a domain expert, used the codebook to code a random sample of 200 pull requests. This analysis spanned four sessions, with 50 pull requests coded in each. After each session, we updated the codebook's description of when to use a code and when not to, along with examples, based on coding discrepancies. This process resulted in four versions of the codebook.

The replication package includes detailed metrics on inter-rater reliability and documents the evolution of the codebook.

4. Results

4.1. Description of collected data

Identified repositories. We identified a total of 59 576 repositories that contain configuration files for either Playwright or Cypress and therefore potentially include GUI-based tests. Of these repositories, 67% had no stars, suggesting they are primarily hobby projects, experimental demonstrations, coursework, or early-stage developments.

The most starred repository was `freeCodeCamp`,[8] an open-source curriculum for learning programming, with approximately 451 000 stars. Repositories in the top 100 each have at least 19 000 stars. Among these repositories, 84 had pull requests (PRs) modifying GUI-based test files, and 56 explicitly discussed GUI-based tests.

Pull requests related to GUI-based test files. In total, we gathered data from 878 002 PRs. As shown in Table 3, modifications to GUI-based test files are rare, and code review discussions of these files are rarer still. Only 1.9% (16 866 PRs) of all PRs modify GUI-based test files, and just 0.25% (2192 PRs) contained comments or discussions.

Pull request templates. Pull request templates help standardize contributions by prompting authors to complete specific steps or include required information, often through checklists. By incorporating required information directly into the pull request workflow, these templates effectively operationalize code review guidelines. As with other forms of operationalization, such as automated checks, we assume that repository maintainers added topics to templates to address recurring issues during pull request review.

Code review guidelines can be viewed from the perspective of the reviewer and the author of a change. They help reviewers conduct more effective reviews and help authors prepare their changes for review. This includes providing information that reviewers would otherwise have to request from the author.

In the following section, we will show that some of the previously proposed guidelines are operationalized by the templates.

We decided to collect templates from the entire set of the top 100 repositories rather than restricting it to the 84 repositories that contain GUI-based test files, as the templates in these repositories include questions related to testing, such as a checkbox on whether necessary unit or end-to-end tests were added. Among the top 100 sampled repositories, 71 included templates. Of these 71 templates, 11 are from repositories that do not include GUI-based test files.

4.2. Findings from pull request templates

We analyzed 71 PR templates from 100 repositories and identified recurring elements. The templates generally requested either a text description (e.g., explaining why the change is necessary) or items to be checked from a checklist (e.g., does it contain breaking changes: yes/no). In our report, we only include the elements that appear in templates from at least two repositories to exclude solely project-specific elements.

Links to related resources (53 templates). Contributors are most often asked to link to the relevant GitHub issue, since feature proposals and bug discussions occur there. Some repositories require opening an issue before submitting a pull request to discuss the envisioned changes first, which shifts part of the discussion away from the code review. Templates also frequently request links to specifications, design documents, or other contextual resources, as well as to any dependent PR or issues that must be resolved first.

[8] https://github.com/freeCodeCamp/freeCodeCamp.

Table 3
Overview of top 100 repositories and the number of total pull requests, pull requests changing GUI-based test files (w/ tests), and pull requests containing code review comments addressing GUI-based test files (w/ review).

#	Repository	Total	w/ tests	w/ review	#	Repository	Total	w/ tests	w/ review
1	freeCodeCamp/freeCodeCamp	39 974	432	154	51	hasura/graphql-engine	3397	0	0
2	facebook/react	17 280	13	3	52	vercel/swr	964	10	0
3	tailwindlabs/tailwindcss	3 638	0	0	53	remix-run/remix	4 696	0	0
4	storybookjs/storybook	13 837	254	8	54	floating-ui/floating-ui	1 379	68	0
5	n8n-io/n8n	11 666	651	117	55	alan2 207/bulletproof-react	115	4	0
6	supabase/supabase	14 221	8	0	56	cockroachdb/cockroach	73 308	11	0
7	mermaid-js/mermaid	2 911	0	0	57	refinedev/refine	4 838	43	0
8	microsoft/playwright	16 537	57	9	58	ianstormtaylor/slate	2 488	12	2
9	coder/code-server	1 980	66	22	59	backstage/backstage	21 820	2	1
10	louislam/uptime-kuma	1 741	10	3	60	codex-team/editor.js	1 154	124	30
11	grafana/grafana	62 133	752	113	61	fabricjs/fabric.js	3 195	54	18
12	apache/superset	18 373	145	23	62	ueberdosis/tiptap	1 492	0	0
13	immich-app/immich	7 122	41	5	63	nextcloud/server	31 217	366	25
14	facebook/docusaurus	6 433	15	0	64	docsifyjs/docsify	1 146	0	0
15	nuxt/nuxt	7 690	66	11	65	xyflow/xyflow	1 632	9	0
16	gatsbyjs/gatsby	22 330	0	0	66	menloresearch/jan	2 265	18	0
17	remix-run/react-router	4 170	0	0	67	pmndrs/react-spring	603	0	0
18	nocodb/nocodb	5 464	441	16	68	motiondivision/motion	1 160	0	0
19	ionic-team/ionic-framework	8 101	0	0	69	AdguardTeam/AdGuardHome	494	0	0
20	withastro/astro	8 023	386	34	70	GitbookIO/gitbook	1 292	75	5
21	toeverything/AFFiNE	8 686	1099	28	71	twentyhq/twenty	6 714	56	14
22	marktext/marktext	1 075	11	0	72	DioxusLabs/dioxus	1 856	46	0
23	TryGhost/Ghost	15 999	118	4	73	paperless-ngx/paperless-ngx	2 975	32	0
24	laurent22/joplin	5 181	72	10	74	keycloak/keycloak	19 737	81	11
25	cypress-io/cypress	8 518	0	0	75	nextauthjs/next-auth	3 199	4	1
26	slab/quill	784	8	0	76	ZuodaoTech/everyone-can-use…	694	16	0
27	FuelLabs/fuels-ts	2 383	8	3	77	PostHog/posthog	24 958	937	85
28	RocketChat/Rocket.Chat	18 199	728	130	78	eslint/eslint	7 890	0	0
29	dcloudio/uni-app	522	0	0	79	marmelab/react-admin	6 054	204	27
30	tldraw/tldraw	3 660	36	10	80	nrwl/nx	15 453	111	2
31	expo/expo	19 097	46	0	81	goharbor/harbor	9 626	1	0
32	tabler/tabler	1 041	0	0	82	badges/shields	8 145	7	1
33	penpot/penpot	4 312	114	11	83	openai-translator/openai-translator	683	8	1
34	gradio-app/gradio	5 093	74	10	84	Redocly/redoc	764	31	1
35	eugenp/tutorials	17 732	0	0	85	BabylonJS/Babylon.js	13 337	19	3
36	trpc/trpc	4 572	0	0	86	react-navigation/react-navigation	1 950	12	0
37	novuhq/novu	6 478	78	5	87	usablica/intro.js	1 236	7	3
38	appsmithorg/appsmith	17 130	1623	309	88	forem/forem	14 840	453	149
39	ray-project/ray	32 344	60	6	89	plausible/analytics	2 748	37	4
40	slidevjs/slidev	760	12	0	90	remotion-dev/remotion	3 355	0	0
41	Kong/insomnia	3 972	819	66	91	BerriAI/litellm	5 051	18	0
42	logseq/logseq	3 410	183	28	92	QwikDev/qwik	4 962	1	0
43	calcom/cal.com	12 910	775	209	93	facebook/lexical	4 445	616	52
44	ToolJet/ToolJet	7 699	535	19	94	wandb/openui	55	2	0
45	payloadcms/payload	6 394	1917	80	95	handsontable/handsontable	3 010	39	2
46	spacedriveapp/spacedrive	2 073	10	1	96	eclipse-theia/theia	6 595	41	8
47	jaredpalmer/formik	1 348	3	0	97	hcengineering/platform	6 249	489	19
48	usebruno/bruno	1 622	0	0	98	apollographql/apollo-client	6 815	3	1
49	harness/harness	1 527	0	0	99	sveltejs/kit	6 242	573	72
50	mattermost/mattermost	21 665	560	108	100	microsoft/fluentui	19 899	0	0

Summary of changes (45 templates). Many templates require a concise summary of the modifications introduced. Some also mandate a changeset file for version tracking in the release process.

Motivation for the change (27 templates). In addition to the summary of changes, contributors should articulate the rationale: why the change is necessary, what it affects, and which design decisions or trade-offs were taken during its implementation.

Test plan description (24 templates). In one-third of the identified pull request templates, contributors must describe how the changes can be tested. This includes information about the environment configuration, minimal test data, the expected behavior for key scenarios, and any other information that helps to test the pull request.

Visual demonstration (20 templates). When applicable, contributors should include a video, animated GIF, or screenshots that illustrate feature usage or user-interface changes.

Before/after screenshots (10 templates). Some templates specifically request paired screenshots to highlight visual differences in the GUI.

Highlighted changes for reviewers (2 templates). A small number of templates ask contributors to highlight particular code sections or files for reviewers to prioritize their review.

Checklist items. Contributors should check or uncheck items that are presented in Table 4 if applicable.

4.3. Mapping code review comments to GUI-based testing guidelines

Out of the 1000 analyzed PRs that included discussions related to GUI-based testing, 808 instances (80.8%) were aligned with the previously proposed guidelines.

In the following analysis, we detail the level of support observed for each guideline, organized by category. Table 5 and Fig. 3 summarize the evidence supporting previously proposed guidelines based on the gathered data. As evidence, we consider the analyzed code review

Table 4
Checklist items in pull request templates and the number of templates (#) they appear in.

Aspect	#
Documentation was added, updated, or checked if it is still up-to-date	29
Tests were added or updated and cover new code	28
Change type specified, such as bug fix, feature, documentation, refactoring, etc.	14
Project-specific contribution guidelines were read and followed	17
Style guidelines were followed	12
Breaking changes were introduced	8
Tests were executed locally	6
Self-review by the author was performed	6
User-facing (GUI) changes were introduced	4
PR scope is as isolated as possible (only one feature) without unrelated changes	2

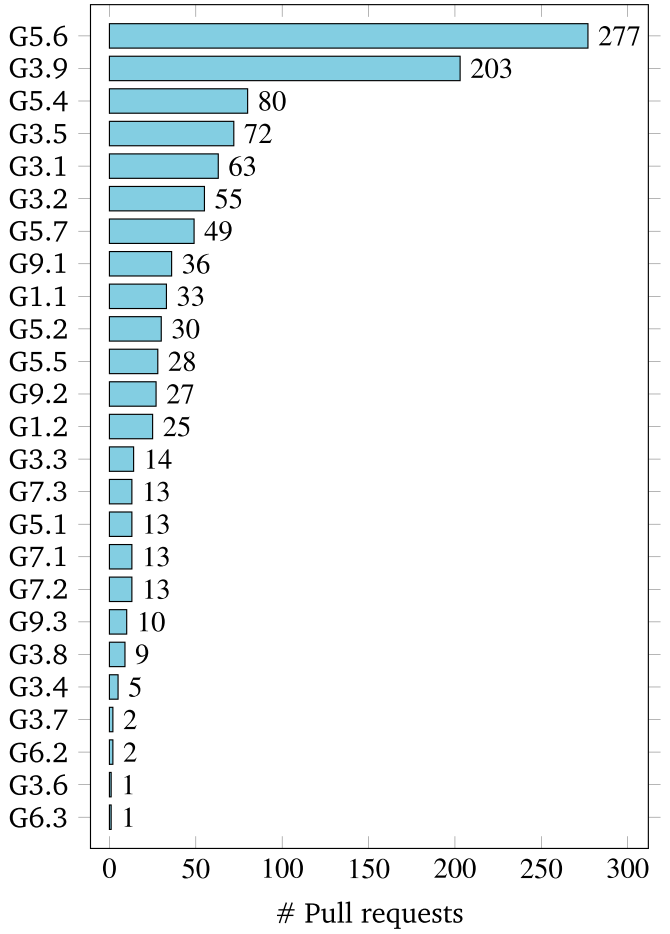

Fig. 3. Guidelines observed in code review comments found in open-source repositories.

comments at the PR granularity level, as well as insights from the PR templates.

G1: Perform automated checks. The previously proposed guidelines for automated checks focus on two main areas: general checks to identify incorrect code patterns and ensure code style conformity. All 84 analyzed repositories employ automated checks via continuous integration (CI) pipelines (e.g., GitHub Actions) or third-party bots that integrate into GitHub.

Examples of automated checks relevant to GUI-based tests include providing test execution results (such as test coverage and execution time) and checking adherence to code style rules. To report GUI coverage, Cypress provides a report that includes a coverage score based on the ratio of interacted elements to all interactable elements available. Playwright allows users to record complete traces, highlighting interactions with GUI elements, for reporting. AI/LLM-based code review bots are also utilized. Finally, the licensing of contributions through a contributor license agreement (CLA) is requested automatically.

From the code review discussion, we noticed that 33 PRs include comments that were related to automated checks (G1.1), pointing out typos that can be easily fixed with tools. Additionally, automated code style checks (G1.2) were mentioned 25 times. These comments included simple adjustments, such as correcting spacing or adding empty lines between functions, which are typically enforced by linters. In some cases, style issues were introduced after merging other branches into one of the PRs, despite a linter being available.

G2: Use checklists. Our use checklist guideline covers aspects such as GUI element localization, test oracles, and test synchronization. While there were no code review comments related to the checklist, 43 of the 71 pull request templates included predefined checks, as detailed in Section 4.2. The items covered by these checklists were generally applicable, but only one was specifically related to testing: it involved verifying whether a new or modified test was successfully executed in a local environment.

G3: Provide context information. Providing context information encompasses a broad category of guidelines aimed at equipping reviewers with additional information to better understand the changes, thus avoiding the need for reviewers to request this information during reviews.

The general guideline of providing context information (G3.1) helps to better understand the change under review. Since all sampled GitHub repositories utilize automated bots, they already provide additional information without specific requests from reviewers. For instance, bots offer links to a deployment or instance of a preview environment for the system under review, provide summaries of changed file sizes, and report the sizes of bundled packages. In 63, reviewers requested more context information to understand a particular aspect of the tests.

Providing a rationale (G3.2) is a common requirement in PR templates. We identified it in 27 templates. In 55 PRs, reviewers requested a rationale for specific tests to clarify changes. Contributors should explicitly state the purpose and motivation, such as why a test case was removed, since these cannot be deduced from the code alone.

Providing information about the impact of a change (G3.3) was observed in 14 PRs. Related discussions addressed the execution time of tests, the effects of parallel execution, and changes needed to prevent test failures or flakiness.

Providing design and architecture details (G3.4) appeared in 14 PRs, where discussions concerned the overall structure of tests and related aspects, such as credential handling.

Linking to related resources (G3.5) can be observed often (72 PRs) as providing links to other test files that serve as implemention references, testing-tool documentation, and related GitHub issues/PRs.

Providing dependency information between test and production code (G3.6) was not observed in the analyzed PRs.

Providing change history (G3.7) occurred in only 2 PRs.

Discussing edge cases (G3.8) appeared in 9 PRs, covering scenarios such as boundary values and inconsistent behavior across browsers.

Prioritizing files under review (G3.9) was present in only two PR templates, but could be observed often in review comments. It was the second most observed guideline, with 203 PRs containing comments by the change contributor to draw reviewers' attention to specific files and sections. In these comments, contributors provide context or ask questions to reviewers.

G4: Utilize metrics. For the guidelines on utilizing metrics, we found support only in automated reports. Test execution and coverage are provided via GitHub Actions or bot comments. There were no discussions on code metrics between reviewers and contributors.

Table 5
Overview of guidelines and the supporting evidence found in open-source repositories.

ID	Description	PRs	Other evidence
G1	**Perform automated checks**		
G1.1	Automated checks	33	GitHub Actions
G1.2	Automated code style checks	25	GitHub Actions
G2	**Use checklists**		
G2.1	Use a checklist	0	43 PR templates
G3	**Provide context information**		
G3.1	Additional context information	63	GitHub Actions
G3.2	Rationale of change	55	27 PR templates
G3.3	Impact of change	14	–
G3.4	Design & architecture	5	–
G3.5	Links to related resources	72	53 PR templates
G3.6	Dependencies between test and production code	1	–
G3.7	History of changes	2	–
G3.8	Edge cases	9	–
G3.9	Prioritization of files	203	2 PR templates
G4	**Utilize metrics**		
G4.1	Measure code metrics	0	GitHub Actions
G4.2	Measure execution time	0	GitHub Actions
G4.3	Measure test coverage	0	GitHub Actions
G5	**Ensure readability**		
G5.1	Ensure readability	13	–
G5.2	Provide code comments	30	–
G5.3	Follow code/naming style (same as G5.4)		see G5.4
G5.4	Follow code/naming style in test files	80	–
G5.5	Avoid code comments	28	–
G5.6	Proper usage of testing techniques and exception handling	277	–
G5.7	Correct assertions	49	–
G6	**Visualize changes**		
G6.1	Visualization of changes	0	–
G6.2	Visualization of the impact of changes	2	–
G6.3	Traceability and easy navigation	1	–
G7	**Reduce complexity**		
G7.1	Keep size of changes low	13	–
G7.2	Keep complexity low	13	–
G7.3	Avoid unrelated changes	13	2 PR templates
G8	**Check conformity with the requirements**		
G8.1	Adherence to requirements	0	–
G9	**Follow design principles and patterns**		
G9.1	Use of design patterns	36	–
G9.2	Do not repeat yourself	27	–
G9.3	Avoid hardcoded values	10	–
G9.4	SOLID principles	0	–

G5: Ensure readability. The guidelines in this category focus on the readability and clarity of test code, addressing reviewers' concerns about code that is confusing or unclear.

Ensure the readability of tests (G5.1) is the general guideline that captures comments that emphasize overall test readability but do not fall under other specific subcategories. In 13 PRs, we could observe the need to improve the readability, where certain test sections were difficult to read and should be refactored.

Adding code comments (G5.2) was requested in 30 PRs to clarify ambiguous code. Examples of the requested comments include explanations for why a variable is set in the test, descriptions of complex locators, reasons for potential errors and how they are handled, as well as deviations from project-specific practices.

Following a coding style in test writing (G5.3 and G5.4[9]) could also be observed in 28 PRs. Adherence to a specific code style and naming convention is generally enforced through automated checks using a linter. However, there were instances where the reviewer requested that the test code be formatted according to the style guide or that variable names align with the project's naming conventions. In many cases, reviewers expressed concerns about the naming of test case descriptions, noting that they often contained typos, were inconsistent with other test case descriptions, or did not clearly convey the intent of the test case. These issues cannot be enforced through automated checks.

Removeing code comments (G5.5) is the inverse of G5.2 and could be observed in 28 PRs. Reviewers requested the removal of outdated or uninformative comments that no longer aid comprehension due to code changes. A significant concern is the presence of commented-out code that is no longer in use. This not only reduces the readability of the test code but also creates confusion for testers and reviewers about the potential relevance of the test code in the future. Reviewers have requested the removal of this commented-out code or provided an explanation of why it should not be deleted. In cases where the code is infrequently used, reviewers recommend employing techniques such as feature flags to enable tests only in specific situations.

The proper usage of testing techniques and exception handling (G5.6) was observed most frequently, appearing in 277 PRs. Beyond readability, G5.6 also relates to maintainability. The use of familiar testing patterns improves reviewer comprehension and supports long-term code quality. Reviewers frequently highlighted problems such as incorrect use of mocks, testing at inappropriate levels of abstraction, or improper variable initialization, factors known to contribute to test unreliability (Spadini et al., 2018).

[9] G5.3 and G5.4 are considered the same in this study. In our previous research, we listed G5.4 as a separate guideline because it was mentioned in the source material specific for testing.

We observed that reviewer feedback focused on the correct use of locators and timeout/wait because reviewers were concerned that an improper use of these techniques would introduce flakiness of tests.

Modern testing frameworks provide various locator methods to identify GUI elements based on their properties. Reviewers often recommended replacing less precise locators with more robust alternatives. For example, in one repository, there was a strong preference for using the `findByRole` locator, which targets elements by semantic role (e.g., buttons) over the more generic `findByText` locators. Examples of locator methods are shown in Listing 1.

```
cy.findByText("confirm").click() // avoid
cy.findByRole('button', { name: /confirm/i }).
  click()
```

Listing 1: Example of two locators in Cypress targeting the same element on the GUI

Explicit wait statements (e.g., `cy.wait(2000)`), to wait for results/responses of the system under test (SUT), are often not necessary and should be avoided. Reviewers have requested the removal of these explicit wait statements. When timeouts need to be defined, reviewers recommend consolidating the timeout settings either globally or for specific test cases, rather than setting them for individual test steps (see Listing 2).

To ensure the correctness of assertions (G5.7), reviewers flagged assertions that were incorrect, unnecessary, or suboptimal. This guideline could be observed in 49 PRs. Unnecessary assertions included those on intermediate steps not relevant to the test objective. Suboptimal assertions included vague checks, such as verifying a button by its label instead of confirming the expected system behavior. In Playwright, the `click()` method automatically waits for and checks the visibility of the targeted element by default. Therefore, additional visibility checks are not needed. In one instance, a reviewer discouraged the use of regular expressions for assertions, stating: "Assertions in our tests should be strict and concise. Using regexes means that we're losing strictness unless your regex is watertight".[10]

```
await page.goto('/home')
await page.waitForTimeout(5000) //avoid
```

Listing 2: Example of an explicit timeout to wait for a transition to another page.

G6: Visualize changes. This set of guidelines addresses additional visualizations of test case changes that aid their comprehension.

Visual graph representations (G6.1) were not observable from either code review comments or PR templates.

Screenshots or recordings (G6.2) appeared in PR templates as before-and-after screenshots and in two PR comments highlighting targeted GUI elements. In both instances, the screenshot showed the rendered GUI and the browser's debug view of the DOM, to highlight the nesting of an element within another one, and to propose an element's attribute to locate it.

Traceability and navigation between files (G6.3) were observed in only one PR. The relevant PR template referred to visual changes in the system under test, not to visual representations of GUI states covered by tests or of test interactions. Given the low number of related comments, support for this guideline category appears to be limited.

G7: Reduce complexity. This category of guidelines concerns reducing the complexity of test file changes.

Keep changes small (G7.1) was observed in 13 PR comments, often by avoiding unnecessary tests or checks.

Keep the complexity low (G7.2) appeared in 13 PR comments. This guideline favors simple solutions, which may conflict with design principles such as Do not Repeat Yourself (G9.2), which would introduce abstractions for better reusability.

Avoiding unrelated changes (G7.3) was observed in two PR templates and 13 PR comments.

When analyzing the number of files changed per pull request, particularly the distribution between production code and test code files, it becomes evident that test case files are often included alongside production code files rather than as separate modifications. For pull requests containing GUI-based test files, the median number of files modified is 8, while the mean is substantially higher at 41. This suggests that updates to GUI-based test cases are typically made in conjunction with feature changes, rather than being addressed independently.

G8: Check conformity with the requirements. We could not find any support in the code review comments that discuss the adherence of the test code to its requirements.

G9: Follow design principles and patterns. This guideline category promotes the following of design principles and patterns. Specific to GUI-based testing is the page-object model (Leotta et al., 2013a), a common abstraction in GUI testing.

Applying design principles and patterns (G9.1) was observed in 36 PRs. Beyond recommending the page-object model, reviewers requested adherence to project-specific design conventions. For instance, as one reviewer commented, "*I would prefer we kept this code portable by using the JS primitive Object.keys and making decisions about unreachable cases right now*".[11]

Do not repeat yourself (G9.2) appeared in 27 PRs, where reviewers encouraged reuse of recurring test steps, complex locators, and assertions.

Avoiding hardcoded values (G9.3) was noted in 10 PRs. Reviewers advocated configuring tests via variables or environment settings, favoring parameterization over hardcoded values. As one reviewer commented, "*Please replace the hard-coded values with fixtures or variables for better maintainability*".[12]

We did not find any references to the SOLID design principles (G9.4) —Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—in the code review comments.

4.4. Reproducibility with independent coder

Inter-coder reliability was assessed between the initial coders and the independent coder across iterative refinement cycles of the codebook. In the first three iterations (three sessions; $N = 150$ coded pull requests), coders used 23 distinct codes grouped into 6 categories, yielding a macro-mean Cohen's κ of 0.18 across codes and 0.26 across categories; agreement in coding intensity (total number of codes assigned per item) was positive but modest (Spearman's $\rho = 0.34$, $p = 1.74 \times 10^{-5}$). After incorporating feedback and continuously improving code definitions and decision rules, reliability increased in the final (4th) iteration ($N = 50$ items), in which coders used 19 distinct codes and 5 categories: macro-mean κ rose to 0.29 across codes and 0.47 across categories, and correspondence in total codes per item strengthened ($\rho = 0.56$, $p = 2.11 \times 10^{-5}$).

Overall, these results indicate that successive codebook revisions were associated with a clear improvement in inter-coder consistency, particularly at the category level.

[10] https://github.com/facebook/lexical/pull/1189.

[11] https://github.com/Kong/insomnia/pull/5076.

[12] https://github.com/appsmithorg/appsmith/pull/34925.

Category-level results were largely conclusive. Categories with higher frequency (G1, G3, G5, and G9) generally achieved moderate to substantial agreement ($\kappa \approx 0.45$–0.68) and parallel moderate–strong correlations in code counts ($\rho \approx 0.47$–$0.66, p < .001$). In contrast, low-prevalence categories (G7) showed poor kappa despite sometimes high raw agreement, reflecting prevalence bias driven by mutual non-use rather than shared positive identifications.

Code-level results were strongly influenced by code prevalence, which was distorted by random sampling and the nature of the data. The prevalent codes (G1.1, G3.5, G3.9, G5.4, G5.6, G5.7, and G9.2), achieved substantial to near-perfect agreement ($\kappa \approx 0.63$–1.00; e.g., several $\kappa \geq 0.78$), typically accompanied by very high observed agreement ($P_0 \approx 0.94$–1.00) and balanced positive identifications between coders.

In contrast, the majority of low-frequency codes showed poor or null agreement ($\kappa \approx 0.00$) and, in several cases, slightly negative kappa ($\kappa \approx -0.06$–0.02), despite often high raw agreement ($P_0 \approx 0.88$–0.98). Here, agreement was driven primarily by mutual non-use ("both no") rather than shared positive identifications, indicating strong prevalence bias. These codes were typically applied by only one coder or only a few times overall, producing inflated P_0 but minimal chance-corrected reliability.

Given that the independent coder is not a domain expert in GUI-based testing, we consider the codebook improvements and coding results a promising baseline for reproducibility efforts.

The replication package contains more detailed information on inter-coder agreement and reliability metrics per code and per category, as well as all iterations of the codebook and the improvements made.

5. Discussion

To answer our research question "*To what extent, and in what ways, do code review comments on real-world projects align with the proposed guidelines for reviewing GUI-based test files (explicitly or implicitly)?*", we discuss our findings from different perspectives.

How well do the proposed guidelines correspond to current practices observed in open-source projects? Current practices observed in open-source projects reflect the proposed guidelines in three ways: (1) implementation via automation, (2) enforcement through pull request templates, and (3) discussion during code review.

First, in the repositories we analyzed, several guidelines are already realized through GitHub Actions and bots. Automated checks (G1), the use of code quality metrics (G4), and aspects of providing contextual information (G3) are already incorporated. This suggests that the principles underlying those guidelines not only could provide benefits, but have already been institutionalized. As a result, incorporating them into new projects is recommended, as automation ensures the application of guidelines, and the effort required to set up automation are low. All analyzed repositories employed GitHub Actions, bots, or both.

Second, some guidelines are reflected in pull request templates. For example, having a checklist for contributions (G2), providing a rationale and a prioritization of files (G3), or avoiding unrelated changes (G7). Although templates do not enforce strict rules, they serve as soft constraints that guide contributors toward consistent submissions ready for review and assist reviewers by providing a structure for the review.

Third, our analysis of review comments shows frequent attention to test readability (G5), particularly in relation to the appropriate use of testing techniques (G5.6) and correct assertions (G5.7). These aspects are often difficult to automate and remain dependent on reviewer expertise. For example, choosing a meaningful GUI locator strategy, avoiding explicit wait/timeout statements (Fulcini et al., 2022), or asserting behavior based on GUI state rather than raw HTML can enhance maintainability and clarity (De Luca et al., 2024). Additionally, the most suitable testing technique will vary depending on the technology stack and the specific project's testing objectives.

Visualizing changes (G6) only appeared in two code review comments. Our previous study found that reviewers often run test cases locally to verify correct interactions with the SUT (Bauer et al., 2025). Providing visualizations of interactions with the SUT could reduce this need by making execution behavior more transparent. However, support for such features depends heavily on the chosen testing framework. For example, Playwright offers the Trace Viewer,[13] a GUI tool that enables detailed inspection of recorded traces after test execution, facilitating debugging of failures in CI. Playwright thus illustrates how visualization can address the need to provide visualizations manually, and we assume similar capabilities may be adopted by other tools.

Do the findings indicate an implicit adoption of the guidelines? Our investigation demonstrates extensive coverage of the underlying principles behind the guidelines in nearly all observed repositories. However, some guidelines (G3.6, G3.7, G4, G6, G8, and G9.4) demonstrate comparatively low coverage in code review comments. The overall coverage of many guidelines indicates that the proposed guidelines represent relevant principles in practice. Thus, we argue that an implicit adoption of these guidelines in practice can be inferred.

How can we interpret the absence of observations for certain guidelines? In this paragraph, we seek to encourage future research into the reasons behind the absence of observations and therefore defer empirical investigation of the underlying factors to future work.

Dependencies between test code and the production code it tests (G3.6) did not appear in review comments, and no clear reason could be identified for this absence. One possible explanation is that when production code and the corresponding test code are part of the same change, the overall summary of changes may be sufficient for reviewers. Further, a reviewer's familiarity with the codebase can have an impact on this guideline, but we cannot assess this based on the data we have gathered. This omission is notable, as reviewers frequently reference related tests or documentation (G3.5). Prior work, including our study (Bauer et al., 2025) and that of Spadini et al. (2018), identifies such dependencies as a recurring information need for reviewers.

For guideline on supplying a history of changes (G3.7) we can assume that the availability of a full commit history and references to prior PRs by GitHub and Git already covers the need described by the guideline. Therefore, this information does not need to be provided or requested, as it is always available for review.

One possible explanation for the lack of comments regarding adherence to requirements (G8.1) is that new contributions to the repository, such as new features and their corresponding tests, should first be discussed in GitHub issues or other platforms before being submitted as pull requests (PRs), as outlined in Section 4.2. Thus, potential discussions regarding requirements are often resolved in alternative ways.

The SOLID principles (G9.4) appear less relevant for GUI-based tests, such as those implemented with Playwright or Cypress. These tests primarily consist of linear sequences of user interactions (e.g., clicking, typing, navigating), which emphasize procedural flows rather than modular design. While additional helper functions used in GUI-based tests could incorporate concepts like inheritance hierarchies or dependency inversion, we did not observe this in the analyzed repositories. As a result, we did not find evidence of SOLID principles being applied in this context, and thus, their practical value for GUI-based tests remains uncertain.

Should the proposed guidelines be refined in light of our empirical observations? Although observed in 203 PRs, comments on file prioritization (G3.9) exhibit a cohesive scope, suggesting no refinement is needed. In contrast, the guideline on testing techniques and exception handling (G5.6) appeared most frequently, with 277 PRs. Given that this guideline is specific to GUI-based testing without a counterpart in production code review guidelines and the large number of diverse examples support refining G5.6, we propose the following fine-grained sub-guidelines, as shown in Table 6.

[13] https://playwright.dev/docs/trace-viewer.

Table 6
Sub-guidelines for G5.6 and the number of pull requests (PRs) where they were observed.

ID	Guideline	PRs
G5.6.1	Select robust locators, like role-based or test-specific locators	100
G5.6.2	Avoid explicit waits and define timeouts on a test-case level	63
G5.6.3	Avoid test annotations `.skip` and `.only`	37

Select robust locators (G5.6.1): According to best practices recommended by Playwright,[14] Cypress,[15] and the Testing Library,[16] GUI elements should preferably be located either by their role or by a dedicated test identifier. Role-based locators (e.g., `page.getByRole()` in Playwright) align with how users and assistive technologies perceive the GUI under test, for instance, as a button, checkbox, or dropdown. In combination with a text attribute, role-based locators become more stable than pure text-based locators. Alternatively, elements can be identified using test-specific attributes (e.g., `page.getByTestId()` in Playwright). According to the tools' best practices, locating elements by generic tags, classes, or IDs is fragile, as these attributes are often subject to change. Test-specific identifiers mitigate this volatility and provide the most resilient way of locating elements. However, they require introducing these test-specific attributes first, and they no longer reflect a user's perception of the GUI.

Avoid explicit waits (G5.6.2): Unnecessary explicit wait statements with arbitrary durations constitute an anti-pattern or test smell (Fulcini et al., 2022). Modern frameworks, such as Playwright and Cypress, automatically handle waiting for events and interactions, eliminating the need for manual waits that can slow down execution time. For instance, they resolve pages or locate dynamically loaded elements without additional delay instructions. When specific timeout behavior is required, it should be defined at the test-case level rather than for individual statements. However, synchronization of tests and the SUT remains a significant challenge in GUI-based testing (Nass et al., 2021).

Avoid test annotations `.skip` and `.only` (G5.6.3): Annotations such as `.skip` and `.only` allow testers to focus the test execution on specific tests or temporarily skip failing tests without needing to delete or modify the test code. However, reviewers have noted the presence of these annotations and have requested their removal. Contributors can use these annotations during test development, but sometimes forget to remove them when submitting a PR for code review.

What is the potential practical value of the guidelines for developers? Test files and GUI-based test files are modified and reviewed less frequently than production code, limiting reviewers' opportunities to develop comparable expertise (Bauer et al., 2025; Spadini et al., 2018). A lack of experience in code reviews has a negative long-term effect on software quality (McIntosh et al., 2016). Consequently, integrating the most relevant guidelines for the specific project context can support less experienced contributors and reviewers in improving the quality of code reviews. As demonstrated by Alegroth et al. (2021), programming guidelines can sometimes increase cognitive load for testers when a large number of guidelines is introduced at once. In line with those results, the proposed guidelines can help reduce the cognitive load on reviewers by providing clear criteria to follow if integrated in a sensible manner. Using automation to prepare and assist in the code review process, as observed in our sampled repositories, could further reduce the required cognitive load, although it will not eliminate it entirely. Continuous integration pipelines should enforce syntactic and structural checks, such as linting and test coverage thresholds. Additionally, pull request templates should encourage authors to provide context, rationale, and validation steps for GUI-based tests. In general, code reviews, when automation is applied in a sensible manner, have a positive impact on the review process in many cases (Badampudi et al., 2023).

[14] https://playwright.dev/docs/locators.
[15] https://docs.cypress.io/app/core-concepts/best-practices.
[16] https://testing-library.com/docs/queries/about.

Table 7
Overlap between best practices in end-to-end web test automation (as identified by Ricca and Stocco, 2021) and previously proposed guidelines.

Best practice	Guideline
Use Continuous Integration (CI)	G1.1
Produce detailed reports	G4.1
Use appropriate naming and code conventions	G5.4
Create robust/proper locators/selectors	G5.6
Manage the synchronization with the app	G5.6
Take/use screenshots	G6.2
Keep the tests atomic and short	G7.1
Use the Page Object Pattern	G9.1
Focus on reusable test code	G9.2

The limited discussion observed for guideline G8.1 may indicate a blind spot during review. In particular, the need for additional context (G3.1) and rationale (G3.2) suggests insufficient understanding of the requirements that motivated the changes. One potential explanation is the use of open-source repositories, which may not be created through a software engineering process with proper requirements elicitation.

Comparison of results to other studies. Ricca and Stocco (2021) surveyed the grey literature to identify best practices in end-to-end web test automation. Several of the practices they identified align closely with our findings from the analysis of code review comments and project guidelines. Table 7 shows this alignment.

Among the technical best practices discussed, the proper handling of synchronization between the web application and the test code was highlighted most frequently. This practice aligns with the previously proposed guideline G5.6, which advises against using explicit waits for events from the system under test. Improper handling of synchronization via waits can not only cause an unnecessary increase in test execution time, but also may introduce flaky behavior due to the waits being non-deterministic (Ricca and Stocco, 2021).

Fulcini et al. (2022) presented guidelines to avoid test smells specific to GUI-based testing. Test smells, which build upon the concept of code smells (Fowler, 2018), refer to patterns in test code that hinder readability, maintainability, and overall quality. Based on their guidelines, they developed a linter to identify test smells. Table 8 illustrates how previously proposed guidelines overlap with theirs. Interestingly, their guidelines, R3 to R7, provide a more detailed set of recommendations for using locators. However, they do not mention role-based locators, which has been requested in many of the code review comments we observed. They found that the most frequently violated guidelines in their sampled GitHub repositories were the use of global variables in test cases, followed by the discouraged use of XPath locators (Clark and DeRose, 1999). While issues related to XPath locators have been noted in other studies (Leotta et al., 2013b), we only encountered them in one pull request comment. Similarly, discussions about global variables were observed in a few pull requests. A potential explanation for this discrepancy is the selection of repositories using Selenium as the testing tool, rather than our choice of Playwright and Cypress. Nevertheless, highlighting test smells automatically would help identify potential maintainability issues in test code during code reviews.

Synthesis. Coming back to our research question "*To what extent, and in what ways, do code review comments on real-world projects align with the proposed guidelines for reviewing GUI-based test files (explicitly or implicitly)?*", real-world code review comments largely mirror the intent of the previous proposed guidelines for reviewing GUI-based tests, mostly implicitly rather than explicitly. Many of the previously proposed guidelines are operationalized through CI automation, bots, or via pull request templates.

Table 8
Overlap between guidelines for GUI-based testing maintenance by Fulcini et al. (2022) and previously proposed guidelines.

Guideline by Fulcini et al. (2022)	Guideline
R2: Keep test cases as simple and short as possible	G7.1
R3: Prefer to use locator by Id, CSS locators and Xpath when not available, in that order	G5.6
R4: Use relative XPath in place of absolute XPath	G5.6
R5: Do not use link locators	G5.6
R6: Use tag locators only for multiple elements	G5.6
R7: Use relative URLs locator instead of absolute ones	G5.6
R8: For elements and variables, use a name that mirrors functionalities	G5.4
R9: Keep names of variables clear to everyone	G5.4
R19: Run linters to detect anti-pattern	G1.1
R20: Adopt Page Object Pattern	G9.1
R21: Avoid Thread.sleep statements by turning fixed-time waits into condition-based ones	G5.6

Practitioners and researchers could interpret the observed (80.8%) alignment as evidence that the previously proposed guidelines largely capture shared, practice-relevant reviewing topics. Note that these alignment percentages do not provide a complete view of guideline-relevant topics, as some are operationalized through automated checks (e.g., syntactic/structural) and templating requirements that authors must fulfill (e.g., context and rationale).

GUI-based test files are reviewed and modified less frequently than production code, reducing reviewers' opportunities to develop comparable expertise in this area. The low numbers do not imply that GUI-based tests are not valuable, but could support our proposition that reviewers require further support to thoroughly review GUI-based tests that will be integrated into the codebase.

6. Threats to validity

We discuss the threats to the validity of this study in terms of construct validity, internal validity, external validity, and reproducibility. Further, we discuss the three types of threats that must be addressed in software repository mining studies according to Vidoni (2022): repository bias, data bias, and information bias.

6.1. Construct validity

A central threat to construct validity is that the absence of review comments does not imply that reviewers neglected aspects covered by guidelines or deemed them irrelevant. This illustrates the broader problem that the absence of evidence is not evidence of absence. To strengthen construct validity, we examined pull request templates and information supplied by automation tools (e.g., GitHub Actions, bots) as complementary data sources. Reviewers are unlikely to comment on aspects already covered by contributors or automated feedback, such as test coverage reports. The available data constrain the extent to which conclusions can be drawn about the value provided by the guidelines. Without a comparative study design, a causal attribution is not feasible. As such, the objective of this manuscript is to gain insights into the presence and use, not to measure the concrete value of the guidelines. A comparative study is envisioned for future work.

6.2. Internal validity

A limitation of this study is the qualitative nature of the code review comment analysis. To mitigate researcher bias, the two main authors jointly mapped comments to guidelines in multiple sessions. Each session was limited to a maximum of 90 min to avoid researcher fatigue from analyzing a set of 1000 pull requests. Since this process was carried out collaboratively, no unresolved inconsistencies arose in the mapping. However, to address the potential risk of inter-coder influence, we documented all conflicting interpretations among the authors in the supplementary material (Bauer and Angermeir, 2026). This documentation serves as a guideline for future applications, including topics that cannot be mapped. For example, while refactoring of test cases should be included in the mapping (G9.1), requests for additional tests should not be associated with any guideline. We deem this risk to be less consequential than the risk of producing an incomplete mapping. We consider this manual mapping a strength relative to automated methods. Another potential threat arises from ambiguous pull request comments. When a comment's intent was unclear, we inspected the associated pull request in GitHub to recover the full context of the change.

Comments that could not be reasonably mapped to any guideline were not forced into a guideline, thereby preserving the integrity of the mapping.

6.3. External validity

Repository bias. This study collects data exclusively from publicly available open-source repositories hosted on GitHub. Alternative platforms such as GitLab or GitBucket are excluded, which introduces a repository bias (Vidoni, 2022). Given GitHub's dominance, with approximately 350 million repositories, this exclusion is unlikely to limit the relevance of our findings when sampling 100 popular repositories for the analysis. GitHub is also the most common source of repositories for MSR studies (Vidoni, 2022). It is important to note that these results may vary for less popular repositories. Additionally, this approach indirectly excludes very new repositories, as they have not yet had the opportunity to gain popularity.

In total, we gathered data from 878 002 pull requests across these 100 popular repositories. This reflects the collaboration among many experienced developers and testers in real-world projects, and we believe that this sample is representative of code review practices in popular GitHub repositories.

Open-source software projects differ significantly from closed-source, commercial systems, making it inappropriate to directly transfer insights from open-source to closed-source environments. Open-source projects are typically developed by distributed teams that rely on asynchronous, text-based communication rather than face-to-face interactions. Contributors to open-source projects may participate infrequently or irregularly since they are not contractually tied to the project through employment agreements (Barcomb et al., 2022, 2019; Lee and Carver, 2017).

Studies by Rigby and Bird (2013) and Bosu et al. (2017) compare aspects of code reviews in open-source and closed-source systems. Generally, both studies found a substantial overlap between the code review processes in these two types of software development. However, a notable difference lies in the relationships between code review participants, which are more important in open-source projects, while knowledge transfer is a key focus in closed-source projects (Bosu et al., 2017). We also view knowledge transfer as an important function of code reviews for GUI-based testing. As a result, findings derived from analyzing open-source repositories may not be applicable to proprietary, closed-source environments, where team organization, development workflows, and quality assurance mechanisms can vary considerably.

Data bias. A data bias occurs when there are threats to the validity of the data extracted from the repositories (Vidoni, 2022). In our study, this threat emerges from the selection of testing tools. We focus on Playwright and Cypress, excluding Selenium. Although Selenium remains prevalent in industry and academic literature, its omission reflects our emphasis on modern frameworks. Tool-specific features may influence the mapping to the previously proposed guidelines.

Based on the selection of tools and our focus on web-based applications, the diversity of programming languages is narrowed down to

JavaScript and TypeScript. While Playwright can be used with other languages, such as Python or Java, our sampled repositories only use JavaScript and TypeScript. This potentially limits the transferability of our results to projects in other languages with different tooling and community practices.

6.4. Reproducibility

To mitigate the risks to reproducibility in the results coded by the two main authors, we invited an independent coder for multiple coding sessions to establish a baseline for the reproducibility of our results. This approach bears two main threats. First, the coder was not a domain expert. While this might artificially lower the inter-coder agreement, we perceive it as a potential lower bound to coding reproducibility, strengthening our confidence in the results. Second, to mitigate the risk of the codebook being shaped by the same independent coder across revisions, the codes and categories remained unchanged. Feedback after each coding session was used to improve the descriptions of when to use the code, and changes to the codebook were documented. Despite the limitations of independent coding, it yielded a refined codebook that other researchers can use to confidently replicate our study.

We provide a replication package containing all scripts used for data collection and a set of intermediate artifacts. This package enables researchers to inspect, verify, and repeat the data-gathering process under the same conditions, thereby increasing transparency. By documenting the exact data collection scripts and preserving the raw outputs of intermediate steps, we reduce ambiguity and mitigate threats to reproducibility.

Data volatility. Temporal factors introduce data volatility, which may threaten both the validity and reproducibility of our results. First, the GitHub API evolves over time, altering query structures, rate limits, and the availability of metadata. Such changes can render past data collection procedures irreproducible or introduce inconsistencies across studies (Kalliamvakou et al., 2016). Second, popular repositories are in continuous development, as contributors frequently merge pull requests, refactor code, and expand test suites. Consequently, metrics such as the number of open pull requests or the size of the test base may differ substantially between two collection points, even if separated by only a few days.

To mitigate these threats to some extent, we preserved snapshots of time-intensive intermediate steps, including the complete crawling of pull requests and their metadata from 100 repositories. These snapshots provide stable reference points that support reproduction and allow researchers to validate our results despite the inherent volatility of the original data source.

7. Future work

The present study demonstrates that the proposed guidelines are aligned with concerns raised in code review comments on GUI-based tests. However, alignment alone does not establish a causal link between adopting guidelines and the outcome of code reviews. Future work should therefore move beyond mapping review comments to guidelines and investigate the validity, use, and effects of the guidelines through complementary empirical designs.

Experimental and longitudinal evaluation. Future research should explore whether adopting the guidelines improves the readability, maintainability, robustness, and reliability of tests, as well as the efficiency and consistency of the review process. Although controlled experiments in industrial settings are difficult, approaches like controlled field experiments, quasi-experimental designs, or longitudinal studies could yield stronger evidence than the observational mapping in this study. For example, future studies could compare review outcomes before and after adopting the guidelines, or evaluate teams that follow the guidelines against those that follow their usual practices.

Interviews with pull request authors and reviewers. A key limitation of this study is that the true intentions behind review comments cannot be fully understood from the text alone. Future research should include interviews with pull request authors and reviewers from the selected projects to determine whether our mapping aligns with their interpretations, thereby strengthening the construct validity of the mapping. Such interviews could also help explain why some guidelines were not observed in review comments, for example, because they were considered irrelevant in the given context, addressed through templates or automation, or handled at another stage of the development process. In addition, this could address whether authors and reviewers already follow any guidelines beyond those enforced through pull request templates.

Analysis of subsequent pull request modifications. Another important direction is to investigate whether and how guideline-aligned review comments lead to subsequent modifications in the test code under review. Such analyses would move beyond identifying concerns raised during review and examine the outcomes associated with those concerns. Future work could track whether a guideline-aligned comment was followed by corresponding changes in the pull request, whether the concern was acknowledged but not addressed, or whether it was ignored. When comments are addressed, subsequent analysis could examine whether the resulting modifications plausibly improve the readability, robustness, maintainability, or reliability of GUI-based tests. This type of longitudinal trace analysis would provide a stronger basis for assessing the practical effectiveness of the guidelines in improving the quality of reviewed tests.

Member checking and expert validation. Future work should also conduct member checking or expert validation with practitioners who have experience reviewing GUI-based tests. For example, experienced reviewers could independently assess a sample of mapped comments and determine whether each comment reflects the guideline to which it was assigned. Such validation would provide an additional check on the interpretation of the mappings and would complement the reproducibility evidence reported in this study.

8. Conclusion

In this study, we collected code review data from 100 repositories to empirically evaluate proposed guidelines for GUI-based test reviews. In total, we gathered data from 878 002 pull requests, of which only 1.9% (16 866 PRs) of all PRs modified GUI-based test files, and just 0.25% (2192 PRs) contained comments or discussions.

We manually analyzed 1000 pull requests, of which 808 included code review comments that either directly or indirectly reflected previously proposed guidelines. We found evidence supporting 25 of the 33 proposed guidelines, drawing insights from review comments, review templates, and GitHub Actions. Notably, guideline G5.6, which relates to locators and the behavior of waiting and timeouts, was the most frequently observed, which we then refined.

Additionally, we provide concrete examples of guidelines for the analyzed testing tools, reducing the ambiguity of the guidelines, along with an overview of which guidelines are supported through automation. Finally, we compare our findings with those from other studies.

Our results suggest that many concerns captured by the proposed guidelines are articulated in practice, indicating practical relevance and coverage for GUI-based test reviews. Thus, we see the guidelines as practice-informed recommendations. Our refinements of guidelines and examples could help teams adopt more consistent review practices and inform tool builders about automatable checks.

CRediT authorship contribution statement

Andreas Bauer: Writing – original draft, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Florian Angermeir:** Writing – original draft, Investigation, Formal analysis, Data curation. **Emil Alégroth:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Seth Anglert:** Resources, Data curation.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used Grammarly and ChatGPT in order to improve and rephrase written paragraphs. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Funding

This work was funded by the KKS foundation through the SERT Research Profile project (research profile grant 2018/010) at Blekinge Institute of Technology.

Declaration of competing interest

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

Appendix. Examples of mapping decisions

To complement Section 3.1, particularly Step 7, this appendix provides illustrative examples of how code review comments were mapped to guidelines. The purpose of these examples is to make the mapping process more transparent by showing how we handled representative cases, including comments for which multiple mappings might seem plausible at first glance.

Some review comments could reasonably be associated with more than one guideline, and the two authors involved in the coding occasionally identified different plausible interpretations. To improve consistency, we documented decisions for recurring ambiguous categories of comments in the results spreadsheet after discussion. These recorded decisions then served as reference points when coding similar comments in later stages of the analysis.

The examples below present the original comment(s), the plausible guideline(s) considered, the rationale for the final decision, and the resulting mapping.

Example A

Comment(s): *"[Reviewer]: There isn't anything wrong with the tests, but splitting them would be more appropriate if we are writing unit tests. Since we are writing end-to-end tests here instead, which are expensive to run, we would want to be a little more mindful of the execution".*[17]

Plausible guideline(s) considered: G7.2 (keep complexity low), G9.1 (use design pattern)

Decision rationale: This comment concerns how tests should be organized. More specifically, it discusses whether tests should be split into smaller units or kept combined, given the cost and purpose of end-to-end execution. One possible interpretation is that splitting tests into smaller units reduces the complexity of each test, which aligns with G7.2. Another possible interpretation is that the comment addresses the broader structuring of tests based on execution cost, scope, and maintainability, which aligns with G9.1.

Final mapping: G9.1. We mapped this comment to G9.1 because its primary focus is the overall design and organization of GUI-based tests in light of project-specific trade-offs, rather than the complexity of an individual test.

[17] Repository: freeCodeCamp, PR: 51962.

Example B

Comment(s): *"Adding parallelism creates an additional dimension which could make any flake harder to find, I suggest re-visiting it when we have more stability".*[18]

Plausible guideline(s) considered: G9.1 (use design pattern), G5.6 (proper usage of testing technique)

Decision rationale: This comment concerns the test execution strategy. Similar comments in the dataset discuss introducing parallel execution, enforcing a specific test order, or ensuring test isolation to avoid side effects. Such comments could be interpreted either as reflecting higher-level design considerations (G9.1) or as addressing the appropriate use of a testing technique (G5.6).

Final mapping: G5.6. We mapped this comment to G5.6 because its main concern is the appropriateness of a specific testing technique in the given context, rather than adherence to a broader design principle.

Example C

Comment(s): *"[Author]: I ended up thinking that it didn't make a lot of sense to bring in that whole library for one utility function. Instead, anticipating we may want to test on URL changes more with cypress, I moved the utility function over and exposed it so it can be used by cypress tests from our e2e package. [Reviewer]: Oh yeah, good point. We don't use any @grafana packages in e2e. Could we use URLSearchParams instead of copying the whole function?".*[19]

Plausible guideline(s) considered: G9.2 (do not repeat yourself), G7.1 (keep size of changes low), G7.2 (keep complexity low)

Decision rationale: The author and reviewer discuss how to avoid duplicating logic across tests. This aligns closely with G9.2. Although the proposed solutions differ, both aim to avoid unnecessary duplication. Guidelines related to reducing complexity (G7.2) could also be considered, because the proposed changes may reduce the size (G7.1) or complexity of (G7.2) individual tests.

Final mapping: G9.2. We mapped these comments to G9.2 because the intention to avoid duplicated code is more specific and central than the secondary effect of reducing test complexity.

Example D

Comment(s): *"[Reviewer]: do people still do page objects or is that not cool anymore? also, is it possible to select by IDs instead of other attributes, or is this the recommended approach with Playwright? [Author]: Lol not sure about page objects. Unfortunately, the way the HTML is structured, there isn't an ID to use as a selector".*[20]

Plausible guideline(s) considered: G3.1 (provide contextual information), G5.6 (proper usage of testing technique)

Decision rationale: This discussion includes a request for additional information, and the author explains why an ID-based selector cannot be used in this case. That aspect could be associated with G3.1. However, the reviewer also explicitly asks whether a better locator strategy is available. The use of robust locators aligns with G5.6.

Final mapping: G5.6. We mapped these comments to G5.6 because the central issue is the choice of locator strategy, even though the discussion also contains explanatory context.

Example E

Comment(s): *"[Reviewer]: Any requirement to skip this case? Kindly share details. [Author]: We are putting this feature behind feature flag and later remove this. That's why the test is being skipped now and once I remove feature flag I will remove the file itself".*[21]

Plausible guideline(s) considered: G5.6 (proper usage of testing technique)

[18] Repository: insomnia, PR: 4480.
[19] Repository: grafana/grafana, PR: 45020.
[20] Repository: coder/code-server, PR: 3133.
[21] Repository: appsmithorg/appsmith, PR: 36940.

Decision rationale: We mapped comments related to avoiding test annotations such as `skip` and `only`, which directly affect whether tests are executed, to G5.6. In this example, the use of a feature flag is discussed as the reason for the temporary use of `skip`. More generally, feature flags can be used to mitigate situations in which such annotations would otherwise be introduced or retained.

Final mapping: G5.6.

Example F

Comment(s): *"[Reviewer]: At this point we should test for keys like "Space" and "Enter", shouldn't we? [Author]: Ok! :)".*[22]

Plausible guideline(s) considered: Not mapped, G5.6 (proper usage of testing technique)

Decision rationale: This comment permits two plausible interpretations. On the one hand, it may be read as a request for additional test cases covering more interaction paths through the GUI. Comments that only request additional tests were not mapped in our analysis. On the other hand, it may be interpreted as suggesting that the test should use interaction methods more appropriate to the behavior under test, which aligns with G5.6.

Final mapping: G5.6. We mapped this comment to G5.6 because we interpreted it primarily as feedback on how the test should be performed, rather than as a general request to add more tests.

Example G

Comment(s): *"[Reviewer]: Is it intentional that we are deleting these E2E tests?".*[23]

Plausible guideline(s) considered: G3.1 (additional context information), G3.2 (provide rationale for a change)

Decision rationale: This is a typical comment asking why a change is necessary, which relates to G3.2. A more extended version of such a comment could include a follow-up discussion, where more information on specific aspects of the test is requested, which would relate to G3.1.

Final mapping: G3.2. We mapped this comment to G3.2 because it most directly aligns with the request for the rationale for a change.

Example H

Comment(s): *"[Reviewer]: Just curious about this line, as I know page.getByTestId('New Request') will wait for the element to exist, why do we need to use isVisible() here? Is there anything that causes the element to be created but cannot be clicked? [Author]: From the trace file screenshot,await page.getByTestId('New Request').dblclick(); clicks on the wrong element. So add a isVisible first to ensure that the newly created element appears on UI before double click".*[24]

Plausible guideline(s) considered: G3.1 (additional context information), G5.6 (proper usage of testing technique)

Decision rationale: This comment requests additional information to understand the reason for the change under review, which aligns with G3.1. At the same time, the discussion also concerns the correct use of locators and interaction conditions, which could support a mapping to G5.6.

Final mapping: G3.1. We mapped this comment to G3.1 because its primary function is to request contextual information needed to understand the code under review. A mapping to G5.6 would also be defensible in this case.

Example I

Comment(s): *"[Reviewer]: Is this change to fix the odd firefox e2e fail? I sent a PR to fix it #12188, but skipped for firefox instead since the tests isn't actually working and testing correctly. (I also updated the test). Maybe we can revert this part and merge that PR instead. [Author]: archive.org is down due to a DDOS, was down all day yesterday, I spend a significant amount of time thinking the tests were really broken! https://archive.org/details/movies I added the smallest mp4 I could create, I promise. [Reviewer]: I know you did, it's slightly smaller than mine, but dare I say mine has more pizzazz. Anyways I don't mind if you want to merge with your fix. I'd still like to get mine in too after to fix the tests though as it wasn't really testing the video playback right (it wasn't ever started in the test so currentTime is always 0)".*[25]

Plausible guideline(s) considered: G3.1 (additional context information), G3.5 (links to related resources), G3.8 (edge cases)

Decision rationale: In the first comment, the reviewer requested more information to determine whether the proposed change would effectively resolve the issue, which aligns with guideline G3.1. Both the reviewer and the author provided links and references (PR #12188) to resources that could help evaluate the change, aligning with guideline G3.5. The discussion centers on failed tests with a specific browser, which can be considered an edge case (G3.8). Additionally, discussing the sizes of the MP4 files used as input for the test further supports an edge case scenario.

Final mapping: We have mapped all three plausible guidelines as they complement each other.

Comments that were not mapped

We did not map comments to guidelines when they addressed the functional behavior of the production code or feature under review, or the setup of the overall execution environment (e.g., CI pipelines, containers, or build scripts), without addressing the design, intent, or quality of the test or another test-related artifact. Likewise, we excluded comments that requested additional tests without providing feedback on the test under review.

Comment: *"Should we add a test when Download File is clicked? Is it possible to check that it downloads?"*[26]

Reason for exclusion: This comment only requests additional tests and does not provide feedback on the quality or design of the test under review.

Comment: *"Does this login utilize a UI-based method? I recall someone mentioning that logging in should be done via API instead of the UI. @ApekshaBhosale, could you provide your input on this?"*[27]

Reason for exclusion: This comment was not mapped because its main concern is the implementation approach for the login process rather than how the login behavior should be tested.

Comment: *"Great test coverage!".*[28]

Reason for exclusion: This comment was not mapped because it praises the tests without providing any insights into why the coverage is great, which could then apply to other tests as well.

Data availability

The datasets generated during and/or analyzed during the current study are published on Zenodo: https://doi.org/10.5281/zenodo.17104676 or Bauer and Angermeir (2026).

[22] Repository: penpot/penpot, PR: 4728.
[23] Repository: apache/superset, PR: 30903.
[24] Repository: Kong/insomnia, PR: 8665.
[25] Repository: withastro/astro, PR: 12177.
[26] Repository: grafana/grafana, PR: 93111.
[27] Repository: appsmithorg/appsmit, PR: 39942.
[28] Repository: grafana/grafana, PR: 63211.

References

Alégroth, Emil, Feldt, Robert, 2017. On the long-term use of visual gui testing in industrial practice: A case study. Empir. Softw. Eng. 22 (6), 2937–2971. http://dx.doi.org/10.1007/s10664-016-9497-6, (ISSN: 1382-3256, 1573-7616).

Alégroth, Emil, Feldt, Robert, Ryrholm, Lisa, 2015. Visual GUI testing in practice: Challenges, problemsand limitations. Empir. Softw. Eng. 20 (3), 694–744. http://dx.doi.org/10.1007/s10664-013-9293-5, (ISSN: 1382-3256, 1573-7616).

Alegroth, Emil, Petersen, Elin, Tinnerholm, John, 2021. A failed attempt at creating guidelines for visual GUI testing: An industrial case study. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation. ICST, IEEE, Porto de Galinhas, Brazil, ISBN: 978-1-72816-836-4, pp. 340–350. http://dx.doi.org/10.1109/ICST49551.2021.00046.

Bacchelli, Alberto, Bird, Christian, 2013. Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, San Francisco, CA, USA, pp. 712–721. http://dx.doi.org/10.1109/ICSE.2013.6606617, ISBN: 978-1-4673-3076-3 978-1-4673-3073-2.

Badampudi, Deepika, Unterkalmsteiner, Michael, Britto, Ricardo, 2023. Modern code reviews—Survey of literature and practice. ACM Trans. Softw. Eng. Methodol. 32 (4), 1–61. http://dx.doi.org/10.1145/3585004, (ISSN: 1049-331X, 1557-7392).

Baker, Richard A., 1997. Code reviews enhance software quality. In: Proceedings of the 19th International Conference on Software Engineering - ICSE '97. ACM Press, Boston, Massachusetts, United States, ISBN: 978-0-89791-914-2, pp. 570–571. http://dx.doi.org/10.1145/253228.253461.

Barcomb, Ann, Stol, Klaas-Jan, Fitzgerald, Brian, Riehle, Dirk, 2022. Managing episodic volunteers in free/libre/open source software communities. IEEE Trans. Softw. Eng. 48 (1), 260–277. http://dx.doi.org/10.1109/TSE.2020.2985093, (ISSN: 0098-5589, 1939-3520, 2326-3881).

Barcomb, Ann, Stol, Klaas-Jan, Riehle, Dirk, Fitzgerald, Brian, 2019. Why do episodic volunteers stay in FLOSS communities? In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, Montreal, QC, Canada, ISBN: 978-1-7281-0869-8, pp. 948–959. http://dx.doi.org/10.1109/ICSE.2019.00100.

Barnett, Mike, Bird, Christian, Brunet, Joao, Lahiri, Shuvendu K., 2015. Helping developers help themselves: Automatic decomposition of code review changesets. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, Florence, Italy, ISBN: 978-1-4799-1934-5, pp. 134–144. http://dx.doi.org/10.1109/ICSE.2015.35.

Bauer, Andreas, Angermeir, Florian, 2026. Replication package: The prevalence of code review guidelines for GUI-based testing in open-source. http://dx.doi.org/10.5281/zenodo.17104676.

Bauer, Andreas, Coppola, Riccardo, Alégroth, Emil, Gorschek, Tony, 2023. Code review guidelines for GUI-based testing artifacts. Inf. Softw. Technol. (ISSN: 09505849) 163, 107299. http://dx.doi.org/10.1016/j.infsof.2023.107299.

Bauer, Andreas, Helmfridsson, Tomas, Alégroth, Emil, Schwarz, Georg-Daniel, 2025. When GUI-based testing meets code reviews. Manuscript submittet to *Software Testing, Verification and Reliability*.

Berner, Stefan, Weber, Roland, Keller, Rudolf K., 2005. Observations and lessons learned from automated testing. In: Proceedings of the 27th International Conference on Software Engineering - ICSE '05. ACM Press, St. Louis, MO, USA, p. 571. http://dx.doi.org/10.1145/1062455.1062556.

Bosu, Amiangshu, Carver, Jeffrey C., Bird, Christian, Orbeck, Jonathan, Chockley, Christopher, 2017. Process aspects and social dynamics of contemporary code review: insights from open source development and industrial practice at microsoft. IEEE Trans. Softw. Eng. 43 (1), 56–75. http://dx.doi.org/10.1109/TSE.2016.2576451, (ISSN: 0098-5589, 1939-3520).

Clark, James, DeRose, Steve, 1999. XML path language (xpath). URL https://www.w3.org/TR/1999/REC-xpath-19991116/.

Cohen, Jason, 2010. Modern code review. In: Making Software: What Really Works, and Why We Believe It, first ed. O'Reilly, ISBN: 978-0-596-80832-7, pp. 329–336.

Coppola, Riccardo, Alégroth, Emil, 2022. A taxonomy of metrics for GUI-based testing research: A systematic literature review. Inf. Softw. Technol. (ISSN: 09505849) 152, 107062. http://dx.doi.org/10.1016/j.infsof.2022.107062.

Coppola, Riccardo, Ardito, Luca, Morisio, Maurizio, Torchiano, Marco, 2020. Mobile testing: New challenges and perceived difficulties from developers of the Italian industry. IT Prof. 22 (5), 32–39. http://dx.doi.org/10.1109/MITP.2019.2942810, (ISSN: 1520-9202, 1941-045X).

Coppola, Riccardo, Morisio, Maurizio, Torchiano, Marco, Ardito, Luca, 2019. Scripted GUI testing of android open-source apps: Evolution of test code and fragility causes. Empir. Softw. Eng. 24 (5), 3205–3248. http://dx.doi.org/10.1007/s10664-019-09722-9, (ISSN: 1382-3256, 1573-7616).

Davila, Nicole, Nunes, Ingrid, 2021. A systematic literature review and taxonomy of modern code review. J. Syst. Softw. (ISSN: 01641212) 177, 110951. http://dx.doi.org/10.1016/j.jss.2021.110951.

De Luca, Marco, Fasolino, Anna Rita, Tramontana, Porfirio, 2024. Investigating the robustness of locators in template-based web application testing using a GUI change classification model. J. Syst. Softw. (ISSN: 01641212) 210, 111932. http://dx.doi.org/10.1016/j.jss.2023.111932.

Dong, Liming, Zhang, He, Yang, Lanxin, Weng, Zhiluo, Yang, Xin, Zhou, Xin, Pan, Zifan, 2021. Survey on pains and best practices of code review. In: 2021 28th Asia-Pacific Software Engineering Conference. APSEC, IEEE, Taipei, Taiwan, ISBN: 978-1-66543-784-4, pp. 482–491. http://dx.doi.org/10.1109/APSEC53868.2021.00055.

Engström, Emelie, Runeson, Per, Skoglund, Mats, 2010. A systematic review on regression test selection techniques. Inf. Softw. Technol. (ISSN: 09505849) 52 (1), 14–30. http://dx.doi.org/10.1016/j.infsof.2009.07.001.

Fagan, M.E., 1976. Design and code inspections to reduce errors in program development. IBM Syst. J. (ISSN: 0018-8670) 15 (3), 182–211. http://dx.doi.org/10.1147/sj.153.0182.

Fagerholm, Fabian, Felderer, Michael, Fucci, Davide, Unterkalmsteiner, Michael, Marculescu, Bogdan, Martini, Markus, Tengberg, Lars Göran Wallgren, Feldt, Robert, Lehtelä, Bettina, Nagyváradi, Balázs, Khattak, Jehan, 2022. Cognition in software engineering: A taxonomy and survey of a half-century of research. ACM Comput. Surv. 54 (11s), 1–36. http://dx.doi.org/10.1145/3508359, (ISSN: 0360-0300, 1557-7341).

Fatima, Nargis, Nazir, Sumaira, Chuprat, Suriayati, 2020. Knowledge sharing factors for modern code review to minimize software engineering waste. Int. J. Adv. Comput. Sci. Appl. 11 (1), http://dx.doi.org/10.14569/IJACSA.2020.0110160, (ISSN: 21565570, 2158107X).

Fowler, Martin, 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Fulcini, Tommaso, Garaccione, Giacomo, Coppola, Riccardo, Ardito, Luca, Torchiano, Marco, 2022. Guidelines for GUI testing maintenance: A linter for test smell detection. In: Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation. ACM, Singapore Singapore, ISBN: 978-1-4503-9452-9, pp. 17–24. http://dx.doi.org/10.1145/3548659.3561306.

García, Boni, Del Alamo, Jose M., Leotta, Maurizio, Ricca, Filippo, 2024. Exploring browser automation: A comparative study of selenium, cypress, puppeteer, and playwright. In: Bertolino, Antonia, Pascoal Faria, João, Lago, Patricia, Semini, Laura (Eds.), Quality of Information and Communications Technology. vol. 2178, Springer Nature Switzerland, Cham, pp. 142–149. http://dx.doi.org/10.1007/978-3-031-70245-7_10, ISBN: 978-3-031-70244-0 978-3-031-70245-7.

Garousi, Vahid, Afzal, Wasif, Çağlar, Adem, Işık, İhsan Berk, Baydan, Berker, Çaylak, Seçkin, Boyraz, Ahmet Zeki, Yolaçan, Burak, Herkiloğlu, Kadir, 2020. Visual GUI testing in practice: an extended industrial case study. http://dx.doi.org/10.48550/ARXIV.2005.09303.

Garousi, Vahid, Felderer, Michael, 2016. Developing, verifying, and maintaining high-quality automated test scripts. IEEE Softw. 33 (3), 68–75. http://dx.doi.org/10.1109/MS.2016.30, (ISSN: 0740-7459, 1937-4194).

Garousi, Vahid, Mäntylä, Mika V., 2016. A systematic literature review of literature reviews in software testing. Inf. Softw. Technol. (ISSN: 09505849) 80, 195–216. http://dx.doi.org/10.1016/j.infsof.2016.09.002.

Gojare, Satish, Joshi, Rahul, Gaigaware, Dhanashree, 2015. Analysis and design of selenium WebDriver automation testing framework. Procedia Comput. Sci. (ISSN: 18770509) 50, 341–346. http://dx.doi.org/10.1016/j.procs.2015.04.038.

Gonçales, Lucian José, Farias, Kleinner, Da Silva, Bruno C., 2021. Measuring the cognitive load of software developers: An extended systematic mapping study. Inf. Softw. Technol. (ISSN: 09505849) 136, 106563. http://dx.doi.org/10.1016/j.infsof.2021.106563.

Gonçales, Lucian, Farias, Kleinner, Da Silva, Bruno, Fessler, Jonathan, 2019. Measuring the cognitive load of software developers: A systematic mapping study. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, Montreal, QC, Canada, ISBN: 978-1-7281-1519-1, pp. 42–52. http://dx.doi.org/10.1109/ICPC.2019.00018.

Gonçalves, Pavlína Wurzel, Fregnan, Enrico, Baum, Tobias, Schneider, Kurt, Bacchelli, Alberto, 2020. Do explicit review strategies improve code review performance? In: Proceedings of the 17th International Conference on Mining Software Repositories. ACM, Seoul Republic of Korea, ISBN: 978-1-4503-7517-7, pp. 606–610. http://dx.doi.org/10.1145/3379597.3387509.

Gonçalves, Pavlína Wurzel, Fregnan, Enrico, Baum, Tobias, Schneider, Kurt, Bacchelli, Alberto, 2022. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. Empir. Softw. Eng. 27 (4), 99. http://dx.doi.org/10.1007/s10664-022-10123-8, (ISSN: 1382-3256, 1573-7616).

Gousios, Georgios, Pinzger, Martin, van Deursen, Arie, 2014. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering. ACM, Hyderabad India, ISBN: 978-1-4503-2756-5, pp. 345–355. http://dx.doi.org/10.1145/2568225.2568260.

Greif, Sacha, 2024. State of JS 2024 survey. URL https://2024.stateofjs.com/en-US/libraries/testing/.

Haas, Roman, Elsner, Daniel, Juergens, Elmar, Pretschner, Alexander, Apel, Sven, 2021. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, Athens Greece, ISBN: 978-1-4503-8562-6, pp. 1281–1291. http://dx.doi.org/10.1145/3468264.3473922.

Hassan, Ahmed, 2006. Mining software repositories to assist developers and support managers. In: 2006 22nd IEEE International Conference on Software Maintenance. IEEE, Philadelphia, PA, USA, (ISSN: 1063-6773) ISBN: 978-0-7695-2354-5, pp. 339–342. http://dx.doi.org/10.1109/ICSM.2006.38.

Itkonen, Juha, Mantyla, Mika V., Lassenius, Casper, 2009. How do testers do it? an exploratory study on manual testing practices. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE, Lake Buena Vista, FL, USA, ISBN: 978-1-4244-4842-5, pp. 494–497. http://dx.doi.org/10.1109/ESEM.2009.5314240.

Kalliamvakou, Eirini, Gousios, Georgios, Blincoe, Kelly, Singer, Leif, German, Daniel M., Damian, Daniela, 2016. An in-depth study of the promises and perils of mining GitHub. Empir. Softw. Eng. 21 (5), 2035–2071. http://dx.doi.org/10.1007/s10664-015-9393-5, (ISSN: 1382-3256, 1573-7616).

Kruchten, Philippe, Nord, Robert L., Ozkaya, Ipek, 2012. Technical debt: From metaphor to theory and practice. IEEE Softw. (ISSN: 0740-7459) 29 (6), 18–21. http://dx.doi.org/10.1109/MS.2012.167.

Layzell, P., Brereton, O.P., French, A., 2000. Supporting collaboration in distributed software engineering teams. In: Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000, IEEE Comput. Soc, Singapore, ISBN: 978-0-7695-0915-0, pp. 38–45. http://dx.doi.org/10.1109/APSEC.2000.896681.

Lee, Amanda, Carver, Jeffrey C., 2017. Are one-time contributors different? A comparison to core and periphery developers in FLOSS repositories. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, IEEE, Toronto, ON, ISBN: 978-1-5090-4039-1, pp. 1–10. http://dx.doi.org/10.1109/ESEM.2017.7.

Leitner, Andreas, Ciupa, Ilinca, Meyer, Bertrand, Howard, Mark, 2007. Reconciling manual and automated testing: the AutoTest experience. In: 2007 40th Annual Hawaii International Conference on System Sciences. HICSS'07, IEEE, Waikoloa, HI, ISBN: 978-0-7695-2755-0, http://dx.doi.org/10.1109/HICSS.2007.462, 261a–261a.

Leotta, Maurizio, Clerissi, Diego, Ricca, Filippo, Spadaro, Cristiano, 2013a. Improving test suites maintainability with the page object pattern: An industrial case study. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. IEEE, Luxembourg, Luxembourg, pp. 108–113. http://dx.doi.org/10.1109/ICSTW.2013.19, ISBN: 978-0-7695-4993-4 978-1-4799-1324-4.

Leotta, Maurizio, Clerissi, Diego, Ricca, Filippo, Tonella, Paolo, 2013b. Capture-replay vs. Programmable web testing: An empirical assessment during test case evolution. In: 2013 20th Working Conference on Reverse Engineering. WCRE, IEEE, Koblenz, Germany, ISBN: 978-1-4799-2931-3, pp. 272–281. http://dx.doi.org/10.1109/WCRE.2013.6671302.

Leotta, Maurizio, García, Boni, Ricca, Filippo, Whitehead, Jim, 2023. Challenges of end-to-end testing with selenium WebDriver and how to face them: A survey. In: 2023 IEEE Conference on Software Testing, Verification and Validation. ICST, IEEE, Dublin, Ireland, ISBN: 978-1-66545-666-1, pp. 339–350. http://dx.doi.org/10.1109/ICST57152.2023.00039.

Mantyla, M.V., Lassenius, C., 2009. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. (ISSN: 0098-5589) 35 (3), 430–448. http://dx.doi.org/10.1109/TSE.2008.71.

Matthews, Gerald, Wohleber, Ryan W., Lin, Jinchao, 2019. Stress, skilled performance, and expertise: Overload and beyond. In: Ward, Paul, Maarten Schraagen, Jan, Gore, Julie, Roth, Emilie M. (Eds.), The Oxford Handbook of Expertise, first ed. Oxford University Press, pp. 490–524. http://dx.doi.org/10.1093/oxfordhb/9780198795872.013.22, ISBN: 978-0-19-879587-2 978-0-19-183705-0.

McIntosh, Shane, Kamei, Yasutaka, Adams, Bram, Hassan, Ahmed E, 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21, 2146–2189.

Memon, Atif Muhammed, 2001. A Comprehensive Framework for Testing Graphical User Interfaces. University of Pittsburgh.

Minas, Randall K., Kazman, Rick, Tempero, Ewan, 2017. Neurophysiological impact of software design processes on software developers. In: Schmorrow, Dylan D., Fidopiastis, Cali M. (Eds.), Augmented Cognition. Enhancing Cognition and Behavior in Complex Human Environments. vol. 10285, Springer International Publishing, Cham, pp. 56–64. http://dx.doi.org/10.1007/978-3-319-58625-0_4, ISBN: 978-3-319-58624-3 978-3-319-58625-0.

Na, Kwan-Sik, Li, Xiaotong, Simpson, James T., Kim, Ki-Yoon, 2004. Uncertainty profile and software project performance: A cross-national comparison. J. Syst. Softw. (ISSN: 01641212) 70 (1–2), 155–163. http://dx.doi.org/10.1016/S0164-1212(03)00014-1.

Nass, Michel, Alégroth, Emil, Feldt, Robert, 2021. Why many challenges with GUI test automation (will) remain. Inf. Softw. Technol. (ISSN: 09505849) 138, 106625. http://dx.doi.org/10.1016/j.infsof.2021.106625.

Niranjanamurthy, M, Navale, Sushanth, Jagannatha, S, Chakraborty, Sudesha, 2018. Functional software testing for web applications in the context of industry. J. Comput. Theor. Nanosci. (ISSN: 1546-1955) 15 (11), 3398–3404. http://dx.doi.org/10.1166/jctn.2018.7632.

Pascarella, Luca, Spadini, Davide, Palomba, Fabio, Bruntink, Magiel, Bacchelli, Alberto, 2018. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact. (ISSN: 2573-0142) 2 (CSCW), 1–27. http://dx.doi.org/10.1145/3274404.

Randel, A.E., Jaussi, K.S., 2003. Functional background identity, diversity, and individual performance in cross-functional teams. Acad. Manag. J. 46 (6), 763–774. http://dx.doi.org/10.2307/30040667, (ISSN: 0001-4273, 1948-0989).

Ricca, Filippo, Stocco, Andrea, 2021. Web test automation: Insights from the grey literature. In: Bureš, Tomáš, Dondi, Riccardo, Gamper, Johann, Guerrini, Giovanna, Jurdziński, Tomasz, Pahl, Claus, Sikora, Florian, Wong, Prudence W.H. (Eds.), SOFSEM 2021: Theory and Practice of Computer Science. vol. 12607, Springer International Publishing, Cham, pp. 472–485. http://dx.doi.org/10.1007/978-3-030-67731-2_35, ISBN: 978-3-030-67730-5 978-3-030-67731-2.

Rigby, Peter C., Bird, Christian, 2013. Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, Saint Petersburg Russia, ISBN: 978-1-4503-2237-9, pp. 202–212. http://dx.doi.org/10.1145/2491411.2491444.

Sadowski, Caitlin, Söderberg, Emma, Church, Luke, Sipko, Michal, Bacchelli, Alberto, 2018. Modern code review: A case study at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ACM, Gothenburg Sweden, ISBN: 978-1-4503-5659-6, pp. 181–190. http://dx.doi.org/10.1145/3183519.3183525.

Sánchez-Gordón, Mary-Luz, Colomo-Palacios, Ricardo, 2018. From certifications to international standards in software testing: mapping from ISQTB to ISO/IEC/IEEE 29119-2. In: European Conference on Software Process Improvement. Springer, pp. 43–55.

Savic, Marko, Mantyla, Mika, Claes, Maelick, 2022. Win GUI crawler: A tool prototype for desktop GUI image and metadata collection. In: 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, Valencia, Spain, ISBN: 978-1-6654-9628-5, pp. 223–228. http://dx.doi.org/10.1109/ICSTW55395.2022.00046.

Spadini, Davide, Aniche, Maurício, Storey, Margaret-Anne, Bruntink, Magiel, Bacchelli, Alberto, 2018. When testing meets code review: Why and how developers review tests. In: Proceedings of the 40th International Conference on Software Engineering. ACM, Gothenburg Sweden, ISBN: 978-1-4503-5638-1, pp. 677–687. http://dx.doi.org/10.1145/3180155.3180192.

Spadini, Davide, Palomba, Fabio, Baum, Tobias, Hanenberg, Stefan, Bruntink, Magiel, Bacchelli, Alberto, 2019. Test-driven code review: An empirical study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, Montreal, QC, Canada, ISBN: 978-1-72810-869-8, pp. 1061–1072. http://dx.doi.org/10.1109/ICSE.2019.00110.

Stol, Klaas-Jan, Fitzgerald, Brian, 2018. The ABC of software engineering research. ACM Trans. Softw. Eng. Methodol. 27 (3), 1–51. http://dx.doi.org/10.1145/3241743, (ISSN: 1049-331X, 1557-7392).

Tao, Yida, Dang, Yingnong, Xie, Tao, Zhang, Dongmei, Kim, Sunghun, 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. ACM, Cary North Carolina, ISBN: 978-1-4503-1614-9, pp. 1–11. http://dx.doi.org/10.1145/2393596.2393656.

Vidoni, M., 2022. A systematic process for mining software repositories: Results from a systematic literature review. Inf. Softw. Technol. (ISSN: 09505849) 144, 106791. http://dx.doi.org/10.1016/j.infsof.2021.106791.