The Importance of Code Reviews

Best Practices for Giving and Receiving Feedback

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Understanding Code Reviews

Code reviews are a critical practice in software development, where developers systematically examine each other's code for errors, adherence to coding standards, and overall quality. This practice ensures that the codebase remains maintainable, efficient, and secure while promoting knowledge sharing and team collaboration.

By definition, a code review is a systematic examination of computer source code intended to find and fix mistakes overlooked in the initial development phase, improving the overall quality of the software. The primary goals of a code review are to:

- Identify bugs, errors, or areas for optimization.
- Ensure the code complies with the team's coding standards and best practices.
- Spread knowledge about the codebase and foster a collaborative learning environment among team members.
- Promote discussions and feedback that can lead to better design decisions and more maintainable code.

Types of Code Reviews

Code reviews can take several forms, each with its own set of practices, benefits, and challenges. The main types of code reviews are formal code reviews, informal code reviews, and pair programming.

Formal Code Reviews

Formal code reviews are structured and systematic processes often involving multiple reviewers and several stages. They are thorough and detailed, making them suitable for critical code components and large projects.

- Formal code reviews typically follow a predefined process that includes planning, preparation, review meeting, rework, and follow-up.
- Usually involve multiple reviewers, including senior developers, architects, and sometimes even stakeholders from other departments.
- Each step is documented, and a detailed review report is generated.
- Specialized tools like Review Board, Crucible, or GitHub's pull request system can be used to facilitate the process.

Advantages:

- High thoroughness and detail.
- Comprehensive feedback covering multiple aspects of the code.
- Structured approach ensures that critical issues are less likely to be overlooked.

Challenges:

- Time-consuming and resource-intensive.
- Can be intimidating for junior developers.
- Requires significant coordination and scheduling.

Informal Code Reviews

Informal code reviews are less structured than formal reviews and are often performed ad-hoc. They are quicker and more flexible, making them suitable for smaller changes or when quick feedback is needed.

- The process is more casual and can vary widely. It may involve simple over-the-shoulder reviews or lightweight discussions using pull requests.
- Typically involves fewer participants, often just the author and one or two peers.
- Less formal documentation; feedback is usually provided directly within the code or via comments in the version control system.
- Commonly uses tools like GitHub, GitLab, or Bitbucket for inline comments and discussions.

Advantages:

- Faster and more flexible than formal reviews.
- Encourages frequent and continuous feedback.
- Less intimidating and easier to schedule.

Challenges:

- Potentially less thorough and detailed.
- May miss broader architectural or design issues.
- Dependence on the reviewers' discipline and diligence.

Pair Programming

Pair programming is a collaborative approach where two developers work together at a single workstation, continuously reviewing each other's code as it is written. This type of review is highly interactive and immediate.

- One developer (the driver) writes the code while the other (the observer or navigator) reviews each line as it is written. Roles can switch frequently.
- Involves only two developers working together in real-time.
- Minimal formal documentation, as the review happens live. However, notes and action items can be recorded as needed.
- Can be done using any development environment. Remote pair programming might use tools like Visual Studio Live Share, TeamViewer, or screen sharing services.

Advantages:

- Immediate feedback and quick issue resolution.
- Promotes knowledge transfer and collaboration.
- Can lead to higher code quality due to continuous peer review.

Challenges:

- Can be exhausting and requires good interpersonal skills.
- Potentially less productive for individual tasks.
- May require careful matching of partners to ensure effectiveness.

Why Code Reviews are Crucial

Code reviews play a vital role in the software development lifecycle by ensuring that the code meets the required quality standards, fosters a culture of learning and collaboration, and helps identify issues early in the process.

Improving Code Quality

Adherence to Standards

Code reviews ensure that code follows the team's coding standards and best practices. This consistency makes the codebase easier to read, maintain, and extend. Adherence to design patterns, naming conventions, and documentation standards is checked during reviews, promoting uniformity across the project.

Detection of Errors and Defects

Reviews help identify syntax errors, logical errors, and potential performance issues that automated tests might miss. Reviewers can spot areas where the code can be optimized for better performance and efficiency.

Maintainability

Reviewers often identify complex or convoluted code that may be difficult to maintain or understand. Simplifying such code improves maintainability. Encourages the use of clear and meaningful comments, improving the readability and future maintainability of the code.

Knowledge Sharing

Spreading Expertise

Code reviews provide an opportunity for less experienced developers to learn from their more experienced peers. This sharing of knowledge helps elevate the overall skill level of the team. Reviewers often share insights, alternative approaches, and best practices, contributing to the continuous learning of all team members.

Understanding the Codebase

Regular participation in code reviews ensures that team members are familiar with different parts of the codebase, reducing dependency on individual developers. Facilitates the onboarding of new team members by exposing them to various code segments and the rationale behind design choices.

Cross-Disciplinary Learning

Reviews often involve feedback on not just the code but also related areas such as database design, security considerations, and user interface design. Promotes a well-rounded understanding of the entire application and its ecosystem.

Catching Bugs Early

Preemptive Issue Resolution

Code reviews act as an early warning system to catch bugs before they make their way into the main codebase, reducing the cost and effort required to fix them later. Identifying issues early in the development cycle prevents the accumulation of technical debt and avoids the domino effect of errors compounding over time.

Multiple Perspectives

Different reviewers bring diverse perspectives and expertise, increasing the likelihood of catching subtle bugs and potential issues. Peer review can uncover edge cases and scenarios that the original developer may not have considered.

Testing and Validation

Reviewers often suggest additional tests or validation steps that the original developer might have overlooked, ensuring thorough testing coverage. Encourages writing more testable and modular code, which improves the overall robustness of the application.

Enhancing Team Collaboration

Building a Collaborative Culture

Code reviews foster a culture of collaboration and collective ownership of the codebase. This shared responsibility encourages team members to take pride in their work and contribute to the success of the project. Promotes open communication and constructive feedback, which are essential for a healthy team dynamic.

Improving Communication Skills

Developers improve their ability to articulate technical concepts and provide constructive feedback. This skill is invaluable for team meetings, documentation, and cross-team collaborations. Encourages clear and respectful communication, which is crucial for maintaining a positive and productive work environment.

Conflict Resolution

Regular code reviews can preempt and resolve conflicts related to code design and implementation. By discussing and agreeing on best practices and standards during reviews, teams can avoid misunderstandings and disagreements later. Facilitates a more democratic and inclusive decision-making process, as all team members have a voice in the review process.

The Code Review Process

The code review process is a systematic approach to examining code changes to ensure quality, consistency, and functionality. A well-defined code review process helps teams maintain high standards and promotes effective collaboration.

The code review process generally follows a structured workflow, ensuring that code changes are thoroughly examined and vetted before being integrated into the main codebase. Here's a step-by-step outline of a typical code review workflow:

Preparation

- The author (developer) completes the coding task, writes necessary tests, and ensures the code meets the team's coding standards.
- The author conducts a self-review to catch obvious errors and improve code quality before requesting a formal review.

Create a Code Review Request

- The author submits the code changes for review, usually through a version control system or code review tool.
- The author includes relevant information, such as the purpose of the changes, related tickets, and any specific areas where feedback is needed.

Review Assignment

- Reviewers are assigned based on their expertise, availability, and familiarity with the codebase. This can be done manually or through automated tools.
- Deadlines for completing the review may be set to ensure timely feedback and avoid bottlenecks.

Conducting the Review

- Reviewers examine the code for correctness, readability, performance, security, and adherence to coding standards.
- Reviewers leave comments and suggestions directly in the code review tool, highlighting issues and recommending improvements.

- Authors and reviewers may engage in discussions to clarify points and resolve any disagreements.

Addressing Feedback

- The author revises the code based on the feedback received and updates the code review request.
- The author responds to reviewers' comments, indicating how each issue was addressed.

Approval and Merging

- Once all feedback is addressed, a final review may be conducted to ensure all issues are resolved.
- If the code meets all requirements, reviewers approve the changes.
- The approved code is merged into the main codebase. Automated tests may be run to validate the integration.

Follow-Up

- Occasionally, a post-merge review may be conducted to verify that the changes did not introduce new issues.
- Teams may hold retrospectives to discuss the code review process and identify areas for improvement.

Preparing for a Code Review

Proper preparation is crucial for a successful code review. By ensuring your code is well-prepared and providing a comprehensive review request, you set the stage for effective feedback and productive collaboration.

Before submitting your code for review, take the following steps to ensure it meets the necessary standards and is easy for reviewers to understand and evaluate.

Write Clear and Concise Code

- Adhere to your team's coding standards and best practices. This includes consistent indentation, naming conventions, and code organization.
- Strive for simplicity in your code. Avoid overly complex or convoluted solutions when a simpler approach is available.
- Break down large functions or classes into smaller, more manageable pieces. This not only makes the code easier to understand but also simplifies the review process.
- Use meaningful variable and function names that clearly convey their purpose.

- Avoid abbreviations and cryptic names that can confuse reviewers.
- Write code with the assumption that the reader is unfamiliar with the context, making it as clear as possible.

Commenting and Documenting Code

- Provide comments to explain the "why" behind complex or non-obvious code sections. Focus on the intent and logic rather than restating what the code does.
- Use comments to clarify assumptions, edge cases, and design decisions that may not be immediately apparent.
- Ensure that all public functions, methods, and classes have appropriate documentation. Include information on parameters, return values, and any exceptions that might be thrown.
- Regularly update comments and documentation to reflect any changes made during development. Outdated documentation can be misleading and counterproductive.

Running Tests and Static Analysis Tools

- Ensure that your code is covered by unit tests. Write tests for all new features and bug fixes, and ensure that existing tests still pass.
- Aim for high test coverage, but prioritize meaningful tests that cover a wide range of scenarios, including edge cases.
- Use static analysis tools to detect common coding errors, security vulnerabilities, and potential performance issues. These tools can catch issues that might be missed during manual reviews.
- Address all warnings and errors reported by static analysis tools before submitting your code for review.
- If your changes impact performance, run performance tests to ensure that your code meets the required performance benchmarks.
- Document any performance improvements or regressions, and provide context in your review request.

Creating a Code Review Request

A well-prepared code review request provides reviewers with the necessary context and information to conduct an effective review. Follow these steps to create a comprehensive code review request:

Providing Context and Background

- Provide a brief summary of the changes made in your code. Highlight the main features, bug fixes, or enhancements.
- Include links to relevant tickets, user stories, or design documents to give reviewers a better understanding of the context.
- Clearly state the purpose of the changes. Explain why the changes were made and what problem they aim to solve.
- Describe any new functionality introduced and how it fits into the overall project.

Highlighting Areas of Concern

- Point out specific areas in your code where you would like focused feedback. This could include complex algorithms, new architectural patterns, or sections where you are unsure about the implementation.
- Highlight any known issues or limitations that reviewers should be aware of.
- Pose specific questions to reviewers about areas where you need input or validation. For example, you might ask if a particular approach is optimal or if there are potential edge cases you haven't considered.

Setting Expectations for Reviewers

- Clearly define the scope of the review. Indicate whether you want a full review of all changes or if the focus should be on specific components or functionality.
- If the changes are part of a larger feature, clarify which parts are ready for review and which are still in progress.
- Prioritize the most critical areas for review. For example, if your changes include a new security feature, emphasize the importance of security-related feedback.
- Indicate any areas that have already been thoroughly tested or reviewed to avoid redundant feedback.
- Suggest a reasonable deadline for the review, keeping in mind the complexity of the changes and the availability of reviewers.
- Let reviewers know your availability for follow-up discussions or clarifications. Provide contact information if necessary.

Best Practices for Giving Code Reviews

Providing effective code reviews is an essential skill for any developer. It not only helps improve the quality of the code but also fosters a culture of collaboration and continuous learning within the team.

Adopting a Constructive and Positive Attitude

Approach the review with the goal of helping your colleague improve their code. Your feedback should be aimed at enhancing the quality of the code and supporting the developer's growth.

- Think of the review as a collaborative effort to produce the best possible outcome for the project.
- Acknowledge the effort that went into writing the code. Starting with positive comments can set a constructive tone for the review.
- Encourage the developer by recognizing good practices and improvements they have made.

Focusing on the Code, Not the Coder

Remember that your comments should address the code, not the individual who wrote it. Avoid making feedback personal.

- Use language that emphasizes the code rather than the coder. For example, instead of saying ”You made a mistake here,” say ”This line can be improved by...”
- Base your feedback on objective criteria such as coding standards, best practices, and functionality requirements.
- Avoid biases and focus on providing impartial and fair feedback.

Balancing Thoroughness and Efficiency

- Ensure you thoroughly review the code for all aspects, including functionality, readability, maintainability, and performance.
- Don't rush through the review. Take the time needed to provide comprehensive feedback.
- Focus on the most critical issues first. Prioritize feedback that has the most significant impact on the code's quality and functionality.
- Avoid nitpicking minor issues that do not significantly affect the code. Instead, provide suggestions for minor improvements without overwhelming the developer.

Reviewing Code Effectively

Understanding the Context and Requirements

- Understand the purpose of the code changes. Read the related tickets, user stories, or documentation to get a clear picture of the requirements.
- If the context is unclear, ask the author for clarification before diving into the review.

- Ensure that the code meets all the specified requirements and addresses the problem it was intended to solve.
- Check if the implementation aligns with the overall design and architecture of the project.

Checking for Functionality, Readability, and Maintainability

- Verify that the code performs the intended functions correctly. Look for edge cases and test the code to ensure it handles all scenarios.
- Check for any potential bugs or logic errors.
- Assess the readability of the code. Well-written code should be easy to understand for other developers.
- Ensure that variable and function names are meaningful and that the code is well-organized and formatted.
- Evaluate the maintainability of the code. Consider whether the code can be easily modified or extended in the future.
- Check for modularity and the use of design patterns that promote maintainability.

Identifying Potential Bugs and Vulnerabilities

- Look for common coding mistakes that can lead to bugs, such as off-by-one errors, incorrect variable initialization, and improper error handling.
- Check for potential security vulnerabilities, such as SQL injection, cross-site scripting (XSS), and buffer overflows.
- Ensure that adequate tests are written for the code. Check for unit tests, integration tests, and any other relevant tests.
- Verify that the tests cover a wide range of scenarios, including edge cases and potential failure points.

Providing Clear and Actionable Feedback

- Provide specific feedback that clearly identifies the issue and its location in the code. Use line numbers or code snippets to illustrate your points.
- Explain why something is an issue and how it can be improved. Provide concrete examples or alternative solutions when possible.
- Offer actionable suggestions that the developer can implement to address the issues. Avoid vague comments that do not provide clear guidance.
- Prioritize feedback based on its impact. Highlight critical issues that need immediate attention and suggest improvements for less critical ones.

Communicating Feedback

Using Respectful and Non-Confrontational Language

- Use polite and respectful language in your comments. Avoid harsh or confrontational tones that can be discouraging.
- Frame your feedback as suggestions rather than commands. For example, say ”Consider refactoring this function to improve readability” instead of ”Refactor this function.”
- Focus on providing constructive feedback that helps the developer improve their code. Avoid negative or overly critical comments.
- Highlight both strengths and areas for improvement. Balance your feedback to ensure it is well-rounded and supportive.

Being Specific and Detailed

- Clearly explain the rationale behind your feedback. Help the developer understand the underlying principles and best practices.
- Use examples and analogies to illustrate complex points and make your feedback more accessible.
- Avoid vague or general comments that do not provide specific guidance. For example, instead of saying ”This code is confusing,” say ”This function is doing multiple things at once. Consider breaking it down into smaller, single-purpose functions.”

Offering Suggestions for Improvement

- Offer practical suggestions for how the code can be improved. Provide specific recommendations and examples of how to implement the changes.
- Share relevant resources, such as documentation, articles, or code snippets, that can help the developer understand and address the issue.
- Use the review as an opportunity to teach and share knowledge. Explain best practices and principles that the developer can apply in future work.
- Encourage developers to ask questions and engage in discussions about the feedback. Foster a collaborative learning environment.

Recognizing and Praising Good Practices

- Recognize and praise good practices and improvements in the code. Positive reinforcement encourages developers to continue following best practices.
- Highlight specific examples of well-written code, effective solutions, or clever optimizations.
- Acknowledge the progress and effort that developers make over time. Celebrate improvements and the adoption of feedback from previous reviews.
- Encourage a culture of continuous improvement and learning within the team.

Best Practices for Receiving Code Reviews

Receiving code reviews is an essential part of the development process that offers opportunities for learning and growth. Approaching code reviews with the right mindset and attitude can significantly enhance their value and foster a collaborative environment.

Adopting a positive and constructive mindset is just as important when receiving feedback on your code. Viewing reviews as collaborative efforts rather than personal critiques helps you grow as a developer and contributes to a more productive team environment.

Being Open to Feedback

- Approach code reviews with an open mind. Understand that feedback is meant to help you improve your code and skills.
- Embrace the feedback process as a regular part of your development routine, not as a one-time event.
- Foster an environment where reviewers feel comfortable providing honest and constructive feedback.
- Show appreciation for the time and effort reviewers invest in examining your code.

Understanding that Reviews are Collaborative, Not Personal

- Remember that the feedback is about the code, not about you as a person. Separate your personal identity from the code you write.
- Understand that reviewers aim to improve the overall quality of the project, not to criticize you personally.
- View code reviews as a collaborative effort to enhance the project. Collaboration leads to better solutions and a stronger codebase.
- Engage in discussions with reviewers to gain different perspectives and insights.

Seeing Reviews as Opportunities for Learning and Growth

- Treat each code review as a learning opportunity. Use feedback to identify areas for improvement and develop your skills.
- Reflect on the feedback you receive and apply it to future work to avoid repeating the same mistakes.
- Use reviews to seek guidance on best practices, design patterns, and coding standards.
- Ask reviewers for recommendations on resources or further reading to deepen your understanding.

Responding to Feedback

How you respond to feedback is critical in ensuring that the review process is effective and leads to meaningful improvements in your code.

Evaluating and Understanding the Feedback

- Carefully read through all the feedback provided. Take the time to understand each comment and its context.
- If feedback is unclear, seek clarification before making changes.
- Prioritize feedback based on its impact on the code's functionality, readability, and maintainability.
- Address critical issues first, followed by less critical suggestions.

Asking Clarifying Questions

- If you don't understand a piece of feedback, don't hesitate to ask the reviewer for more information or examples.
- Engage in constructive discussions to gain a deeper understanding of the feedback and its rationale.
- If you have a different approach or perspective, discuss it with the reviewer. Exploring alternative solutions can lead to better outcomes.
- Be open to feedback, but also feel free to respectfully question and discuss suggestions that may not align with your understanding.

Implementing Changes and Updates

- Implement changes based on the feedback. Ensure that the modifications align with the review comments and improve the code.
- Update any relevant documentation or comments to reflect the changes made.
- After making changes, thoroughly test your code to ensure that it still works as expected and that no new issues have been introduced.
- Run existing tests and add new ones if necessary to cover the changes.

Discussing and Negotiating Where Necessary

- Engage in discussions with reviewers when feedback involves significant design changes or architectural decisions.
- Aim for consensus, but be prepared to negotiate and compromise when necessary.
- If you disagree with a piece of feedback, express your views respectfully. Provide reasoning and evidence to support your perspective.
- Be willing to adapt your approach based on constructive discussions and consensus.

Maintaining a Positive Attitude

Maintaining a positive attitude throughout the code review process ensures a constructive and collaborative environment.

Avoiding Defensiveness

- Avoid becoming defensive when receiving feedback. Recognize that the goal is to improve the code, not to criticize you personally.
- Take a moment to process feedback calmly before responding.
- Understand that constructive criticism is valuable and necessary for growth. View it as an opportunity to learn and improve.
- Appreciate the reviewers' efforts to provide feedback, even if it highlights areas for improvement.

Appreciating Constructive Criticism

- Express gratitude for the feedback, regardless of whether it is positive or highlights areas for improvement.
- Acknowledge the reviewers' time and effort in providing detailed and constructive feedback.
- Approach feedback with a learning mindset. Use it as a tool to enhance your skills and knowledge.
- Recognize that every piece of feedback, positive or negative, contributes to your development as a developer.

Recognizing the Value of Different Perspectives

- Value the different perspectives that reviewers bring. Different backgrounds and experiences can provide unique insights that enhance the code.
- Encourage diverse viewpoints to foster a richer and more comprehensive review process.
- Use the feedback process to build a culture of collaboration and mutual respect within the team.
- Recognize that diverse perspectives lead to more robust solutions and a stronger codebase.

Overcoming Common Challenges

Code reviews can present various challenges, from conflicting opinions to handling large or complex changes. Addressing these challenges effectively is crucial for maintaining a productive and positive review process.

Conflicting Opinions

Conflicting opinions are a natural part of the code review process, as different reviewers may have varying perspectives on the best approach to a problem. Effectively resolving these conflicts is essential for maintaining team cohesion and ensuring high-quality code.

Strategies for Resolving Disagreements

- Encourage open and respectful communication. Allow each party to explain their perspective and reasoning.
- Use face-to-face meetings or video calls for complex disagreements, as these formats can facilitate clearer and more effective communication than written comments alone.
- Identify areas of agreement and build from there. Focus on shared goals, such as code quality and maintainability, to find common ground.
- Be willing to compromise and adapt your approach based on constructive discussions.
- Refer to established coding standards and best practices to guide the discussion. Objective criteria can help resolve subjective disagreements.
- If standards do not cover the issue, consider updating them to address similar conflicts in the future.
- Involve additional team members or a neutral third party to provide a fresh perspective. This can help break deadlocks and bring new insights to the discussion.
- Use peer reviews to validate different approaches and reach a consensus based on broader input.

When to Escalate Issues

- Recognize when a disagreement cannot be resolved through discussion and compromise. Persistent conflicts that impact project timelines or quality may need escalation.
- Escalate issues that involve critical design decisions, architectural changes, or potential security risks.
- Use established escalation protocols to address unresolved conflicts. This may involve consulting a senior developer, team lead, or project manager.
- Document the disagreement and the steps taken to resolve it. Provide this information to the person handling the escalation to ensure they have the full context.

Handling Large or Complex Code Reviews

Large or complex code reviews can be overwhelming and time-consuming. Breaking down the review process and prioritizing critical sections can make these reviews more manageable and effective.

Breaking Down the Review Process

- Break down large code changes into smaller, more manageable chunks. Review each chunk separately to maintain focus and thoroughness.
- Use feature branches to isolate different parts of the code change, allowing for more focused and efficient reviews.
- Conduct incremental reviews as the code is being developed. Regular, smaller reviews can be more manageable and less overwhelming than a single large review.
- Encourage developers to submit code for review early and often, rather than waiting until all changes are complete.
- Involve multiple reviewers for large or complex code changes. Assign specific sections or aspects of the code to different reviewers based on their expertise.
- Coordinate the review process to ensure that all sections are thoroughly examined and that feedback is consolidated.

Prioritizing Critical Sections

- Focus on the most critical sections of the code first, such as those affecting core functionality, performance, or security.
- Prioritize sections that are complex, high-risk, or have a significant impact on the overall project.
- Allocate more time and attention to critical sections, while maintaining a reasonable level of scrutiny for less critical areas.
- Use a risk-based approach to determine the level of review required for different parts of the code.
- Use automated tools to handle routine checks, such as code formatting, linting, and basic static analysis. This allows reviewers to focus on more complex and critical aspects of the code.
- Ensure that automated tools are integrated into the development workflow to catch issues early and reduce the burden on manual reviews.

Ensuring Consistency in Reviews

Consistency in code reviews is essential for maintaining high standards and ensuring that feedback is fair and constructive. Establishing team guidelines and using checklists and templates can help achieve this consistency.

Establishing Team Guidelines and Standards

- Develop and document coding standards that cover style, conventions, and best practices. Ensure that all team members are familiar with these standards.

- Regularly review and update the standards to reflect new best practices and lessons learned from previous reviews.
- Establish guidelines for the code review process, including criteria for approving or rejecting changes, expected response times, and escalation procedures.
- Provide examples of good and bad reviews to illustrate the expected level of detail and tone.
- Provide training for new team members on the code review process and standards. Ensure that they understand the importance of consistency and quality in reviews.
- Offer ongoing training and resources to help team members improve their review skills and stay up-to-date with best practices.

Using Checklists and Templates

- Create checklists to guide reviewers through the review process. Checklists can help ensure that all critical aspects are covered, including functionality, readability, maintainability, and security.
- Customize checklists for different types of code changes, such as new features, bug fixes, or refactoring.
- Use templates for review comments to ensure that feedback is clear, consistent, and actionable. Templates can include sections for summary, detailed comments, and suggested improvements.
- Provide examples of well-structured review comments to guide reviewers in providing constructive feedback.
- Integrate automated tools and scripts to perform routine checks and validations. Automation helps maintain consistency and frees up reviewers to focus on more complex issues.
- Ensure that automated checks are aligned with the team's coding standards and review guidelines.