**Best Practices for Code Review**

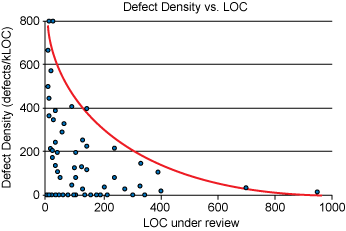
A successful peer review strategy requires balance between strictly documented processes and a non-threatening, collaborative environment. Highly regimented peer reviews can stifle productivity, yet lackadaisical processes are often ineffective. Managers are responsible for finding a middle ground where peer review can be efficient and effective while fostering open communication and knowledge-share between teammates.

**10 tips to guide you toward effective peer code review**

**1. Review fewer than 400 lines of code at a time**

A [SmartBear study of a Cisco Systems programming team](https://smartbear.com/resources/case-studies/cisco-systems-collaborator/) revealed that developers should review no more than 200 to 400 lines of code (LOC) at a time. The brain can only effectively process so much information at a time; beyond 400 LOC, the ability to find defects diminishes.

In practice, a review of 200-400 LOC over 60 to 90 minutes should yield 70-90% defect discovery. So, if 10 defects existed in the code, a properly conducted review would find between seven and nine of them.



**2. Take your time. Inspection rates should under 500 LOC per hour**

It can be tempting to tear through a review, assuming that someone else will catch the errors that you don´t find. However, SmartBear research shows a significant drop in defect density at rates faster than 500 LOC per hour. Code reviews in reasonable quantity, at a slower pace for a limited amount of time results in the most effective code review.

A screen shot of a graph

Description automatically generated

Stay Up to Date on Peer Review Trends

View our interactive report on peer review trends and insights

**3. Do not review for more than 60 minutes at a time**

Just as you shouldn´t review code too quickly, you also should not review for too long in one sitting. When people engage in any activity requiring concentrated effort over a period of time, performance starts dropping off after about 60 minutes. Studies show that taking breaks from a task over a period of time can greatly improve quality of work. Conducting more frequent reviews should reduce the need to ever have to conduct a review of this length.

**4. Set goals and capture metrics**

Before implementing a process, your team should decide how you will [measure the effectiveness of peer review](http://blog.smartbear.com/code-review/measuring-the-benefits-of-peer-review/) and name a few tangible goals.

Using [SMART criteria](https://en.wikipedia.org/wiki/SMART_criteria), start with external metrics. For example, "reduce support calls by 15%," or "cut the percentage of defects injected by development in half." This information should give you a quantifiable picture of how your code is improving. "Fix more bugs" is not an effective goal.

It´s also useful to watch internal process metrics, including:

* **Inspection rate:** the speed with which a review is performed
* **Defect rate:** the number of bugs found per hour of review
* **Defect density:** the average number of bugs found per line of code

Realistically, only automated or strictly controlled processes can provide repeatable metrics. A metrics-driven code review tool gathers data automatically so that your information is accurate and without human bias. To get a better sense of effective code review reporting, you can see [how our code review tool, Collaborator, does it](https://smartbear.com/product/collaborator/features/reporting-and-metrics/).

**5. Authors should annotate source code before the review**

Authors should annotate code before the review occurs because annotations guide the reviewer through the changes, showing which files to look at first and defending the reason behind each code modification. Annotations should be directed at other reviewers to ease the process and provide more depth in context. As an added benefit, the author will often find additional errors before the peer review even begins. More bugs found prior to peer review will yield in lower defect density because fewer bugs exist overall.

**6. Use checklists**

It´s very likely that each person on your team makes the same 10 mistakes over and over. Omissions in particular are the hardest defects to find because it´s difficult to review something that isn´t there. Checklists are the most effective way to eliminate frequently made errors and to combat the challenges of omission finding. Code review checklists also provide team members with clear expectations for each type of review and can be helpful to track for reporting and process improvement purposes.

**Peer Review Checklists :**[**Learn More & Get Examples**](https://smartbear.com/learn/code-review/guide-to-code-review-process/)

**7. Establish a process for fixing defects found**

Even after optimizing code review processes by time-boxing reviews, limiting LOC reviewed per hour and naming key metrics for your team, there´s still a key review step missing. How will the bugs be fixed? It seems obvious, but many teams do not have a systematic method for *fixing* the bugs they´ve worked so hard to find.

The best way to ensure that defects are fixed is to use a collaborative [code review tool](https://smartbear.com/product/collaborator/overview/) that allows reviewers to log bugs, discuss them with the author, and approve changes in the code. Without an automated tool, bugs found in review likely aren´t logged in the team´s usual defect tracking system because they are found before code is released to QA.

**8. Foster a positive code review culture**

Peer review can put strain on interpersonal team relationships. It´s difficult to have every piece of work critiqued by peers and to have management evaluating and measuring defect density in your code. Therefore, in order for peer code review to be successful, it´s extremely important that mangers create a culture of collaboration and learning in peer review.

While it´s easy to see defects as purely negative, each bug is actually an opportunity for the team to improve code quality. Peer review also allows [junior team members to learn from senior leaders](http://blog.smartbear.com/code-review/developing-a-culture-of-mentorship-with-code-review/) and for even the most experienced programmers to break bad habits.

Defects found in peer review are *not* an acceptable rubric by which to evaluate team members. Reports pulled from peer code reviews should never be used in performance reports. If personal metrics become a basis for compensation or promotion, developers will become hostile toward the process and naturally focus on improving personal metrics rather than writing better overall code.

**9. Embrace the subconscious implications of peer review**

The knowledge that others will be examining their work naturally drives people to produce a better product. This "Ego Effect" naturally incentivizes developers to write cleaner code because their peers will certainly see it. The SmartBear study of Cisco Systems found that "spot checking" 20% to 33% of the code resulted in lower defect density with minimal time expenditure. If your code has a 1-in-3 chance of being called out for review, that´s enough of an incentive to double-check your work.

**10. Practice lightweight code reviews**

Between email, over-the-shoulder, Microsoft Word, tool-assisted and hybrids of all types there are countless ways to collaboratively review code. However, to fully optimize your team´s time and to effectively measure its results, a lightweight, tool-assisted process is recommended.

The SmartBear study of Cisco Systems found that **lightweight code review takes less than 20% the time of formal reviews and finds just as many bugs!** Formal, or heavyweight, inspection averages nine hours per 200 LOC. While often effective, this rigid process requires up to six participants and hours of meetings paging through detailed code printouts.