CHAPTER 9

Code Review

第九章 程式碼審查

Written by Tom Manshreck

Caitlin Sadowski Edited by Lisa Carey

Code review is a process in which code is reviewed by someone other than the author, often before the introduction of that code into a codebase. Although that is a simple definition, implementations of the process of code review vary widely throughout the software industry. Some organizations have a select group of “gatekeepers” across the codebase that review changes. Others delegate code review processes to smaller teams, allowing different teams to require different levels of code review. At Google, essentially every change is reviewed before being committed, and every engineer is responsible for initiating reviews and reviewing changes.

程式碼審查是一個由作者以外的人對程式碼進行審查的過程，通常在將該程式碼引入程式碼庫之前。儘管這是一個簡單的定義，但在整個軟體行業中，程式碼審查過程的實施有很大不同。一些組織在程式碼庫中擁有一組挑選出來的 "守門人 "來審查修改。其他人將程式碼審查過程委託給這個小團隊，允許不同的團隊要求不同級別的程式碼審查。在谷歌，基本上每一個改動在提交之前都會被審查，每個工程師都負責啟動審查和審查變更。

Code reviews generally require a combination of a process and a tool supporting that process. At Google, we use a custom code review tool, Critique, to support our process.[1] Critique is an important enough tool at Google to warrant its own chapter in this book. This chapter focuses on the process of code review as it is practiced at Google rather than the specific tool, both because these foundations are older than the tool and because most of these insights can be adapted to whatever tool you might use for code review.

程式碼審查通常需要一個流程和一個支援該流程的工具的組合。在 Google，我們使用一個客製的程式碼審查工具 Critique 來支援我們的流程。 Critique 在 Google 是一個非常重要的工具，足以讓它在本書中佔有一章。本章重點介紹 Google 實施的程式碼審查流程，而不是具體的工具，這是因為這些基礎比工具更古老，而且這些見解大多可以適應你可能用於程式碼審查的任何工具。

Some of the benefits of code review, such as detecting bugs in code before they enter a codebase, are well established[2] and somewhat obvious (if imprecisely measured). Other benefits, however, are more subtle. Because the code review process at Google is so ubiquitous and extensive, we’ve noticed many of these more subtle effects, including psychological ones, which provide many benefits to an organization over time and scale.

程式碼審查的一些好處，例如在程式碼進入程式碼庫之前檢測到程式碼中的錯誤，已經得到了很好的證實，而且有點明顯（如果測量不精確的話）。然而，其他的好處則更為微妙。由於谷歌的程式碼審查過程是如此的普遍和廣泛，我們已經注意到了許多這些更微妙的影響，包括心理上的影響，隨著時間的推移和規模的擴大，會給一個組織帶來許多好處。

> 1 我們也使用 Gerrit 來審查 Git 程式碼，主要用於我們的開源專案。然而，Critique 是谷歌公司典型的軟體工程師的主要工具。
>
> 2 史蒂夫·麥康奈爾, Code Complete (雷蒙德：微軟出版社，2004 年).

Code Review Flow 程式碼審查流程

Code reviews can happen at many stages of software development. At Google, code reviews take place before a change can be committed to the codebase; this stage is also known as a *precommit review*. The primary end goal of a code review is to get another engineer to consent to the change, which we denote by tagging the change as “looks good to me” (LGTM). We use this LGTM as a necessary permissions “bit” (combined with other bits noted below) to allow the change to be committed.

程式碼評審可以發生在軟體開發的多個階段。在谷歌，程式碼評審是在更改提交到程式碼庫之前進行的；這個階段也被稱為*預提交審查*。程式碼評審的主要最終目標是讓另一位工程師同意變更，我們透過將變更標記為“我覺得不錯”（LGTM）來表示。我們將此 LGTM 用作必要的許可權“標識”（與下面提到的其他標識結合使用），以允許提交更改。

A typical code review at Google goes through the following steps:

1. A user writes a change to the codebase in their workspace. This *author* then creates a snapshot of the change: a patch and corresponding description that are uploaded to the code review tool. This change produces a *diff* against the codebase, which is used to evaluate what code has changed.
2. The author can use this initial patch to apply automated review comments or do self-review. When the author is satisfied with the diff of the change, they mail the change to one or more reviewers. This process notifies those reviewers, asking them to view and comment on the snapshot.
3. *Reviewers* open the change in the code review tool and post comments on the diff. Some comments request explicit resolution. Some are merely informational.
4. The author modifies the change and uploads new snapshots based on the feedback and then replies back to the reviewers. Steps 3 and 4 may be repeated multiple times.
5. After the reviewers are happy with the latest state of the change, they agree to the change and accept it by marking it as “looks good to me” (LGTM). Only one LGTM is required by default, although convention might request that all reviewers agree to the change.
6. After a change is marked LGTM, the author is allowed to commit the change to the codebase, provided they *resolve all comments* and that the change is *approved*. We’ll cover approval in the next section.

谷歌的典型程式碼審查過程如下：

1. 使用者在其工作區的程式碼庫中寫入一個變更。然後作者建立變更的快照：一個補丁和相應的描述，上傳到程式碼審查工具。此更改會產生與程式碼庫的*差異*，用於評估已更改的程式碼。
2. 作者可以使用此初始補丁應用自動審查評論或進行自我審查。當作者對變更的差異感到滿意時，他們會將變更郵寄給一個或多個審查者。此過程通知這些審查者，要求他們檢視快照並對其進行評論。
3. *審查者*在程式碼審閱工具中開啟更改，並在差異上發表評論。有些評論要求明確的解決。有些僅僅是資訊性的。
4. 作者根據反饋意見修改修改，並上傳新的快照，然後回覆給審查者。步驟 3 和 4 可重複多次。
5. 在審查員對變更的最新狀態感到滿意後，他們同意變更，並透過將其標記為“我覺得不錯”（LGTM）來接受變更。預設情況下，只需要一個 LGTM，儘管慣例可能要求所有稽核人同意變更。
6. 在更改被標記為 LGTM 之後，作者可以將更改提交到程式碼庫，前提是他們*解決*所有*評論*，並且該變更被*批准*。我們將在下一節中討論批准問題。

We'll go over this process in more detail later in this chapter.

我們將在本章後面更詳細地介紹這個過程。

Code Is a Liability 編碼是一種責任

It's important to remember (and accept) that code itself is a liability. It might be a necessary liability, but by itself, code is simply a maintenance task to someone somewhere down the line. Much like the fuel that an airplane carries, it has weight, though it is, of course, necessary for that airplane to fly.

重要的是要記住（並接受），編碼本身就是一種責任。它可能是一種必要的責任，但就其本身而言，編碼只是某個人的一項維護任務。就像飛機攜帶的燃料一樣，它有重量，儘管它當然是飛機飛行的必要條件。

New features are often necessary, of course, but care should be taken before developing code in the first place to ensure that any new feature is warranted. Duplicated code not only is a wasted effort, it can actually cost more in time than not having the code at all; changes that could be easily performed under one code pattern often require more effort when there is duplication in the codebase. Writing entirely new code is so frowned upon that some of us have a saying: "If you're writing it from scratch, you're doing it wrong!"

當然，新特性通常是必需的，但在開發程式碼之前，首先要注意確保任何新特性都是有必要的。重複的程式碼不僅是一種浪費，而且實際上比根本沒有程式碼要花費更多的時間；當代碼庫中存在重複時，可以在一個程式碼模式下輕鬆執行的更改通常需要更多的工作。編寫全新的程式碼是如此令人不快，以至於我們中的一些人都有這樣一句話：“如果你是從頭開始寫的，那你就是做錯了！”

This is especially true of library or utility code. Chances are, if you are writing a utility, someone else somewhere in a codebase the size of Google's has probably done something similar. Tools such as those discussed in Chapter 17 are therefore critical for both finding such utility code and preventing the introduction of duplicate code. Ideally, this research is done beforehand, and a design for anything new has been communicated to the proper groups before any new code is written.

這對於類別庫或實用程式程式碼來說尤其如此。有可能，如果你正在寫一個實用程式，那麼在像 Google 那樣大的程式碼庫中，其他人可能已經做了類似的事情。因此，像那些在第 17 章中討論的工具對於找到這些實用程式程式碼和防止引入重複的程式碼都是至關重要的。理想情況下，這種研究是事先完成的，在編寫任何新的程式碼之前，任何新的設計都已經傳達給合適的小組。

Of course, new projects happen, new techniques are introduced, new components are needed, and so on. All that said, a code review is not an occasion to rehash or debate previous design decisions. Design decisions often take time, requiring the circulation of design proposals, debate on the design in API reviews or similar meetings, and perhaps the development of prototypes. As much as a code review of entirely new code should not come out of the blue, the code review process itself should also not be viewed as an opportunity to revisit previous decisions.

當然，新專案的發生，新技術的引入，新元件的需要，等等。綜上所述，程式碼審查並不是一個重提或辯論以前的設計決策的時機。設計決策通常需要時間，需要分發設計方案，在 API 評審或類似的會議上對設計進行辯論，也許還需要開發原型。就像對全新的程式碼進行程式碼審查不應該突然出現一樣，程式碼審查過程本身也不應該被看作是重新審視以前決策的時機。

How Code Review Works at Google 谷歌的程式碼審查工作

We've pointed out roughly how the typical code review process works, but the devil is in the details. This section outlines in detail how code review works at Google and how these practices allow it to scale properly over time.

我們已經粗略地指出了典型的程式碼審查過程是如何工作的，但問題在於細節。本節詳細概述了谷歌的程式碼審查工作原理，以及這些實踐如何使其能夠隨時間適當擴充。

There are three aspects of review that require "approval" for any given change at Google:

- A correctness and comprehension check from another engineer that the code is appropriate and does what the author claims it does. This is often a team member, though it does not need to be. This is reflected in the LGTM permissions "bit," which will be set after a peer reviewer agrees that the code "looks good" to them.
- Approval from one of the code owners that the code is appropriate for this particular part of the codebase (and can be checked into a particular directory). This approval might be implicit if the author is such an owner. Google's codebase is a tree structure with hierarchical owners of particular directories. (See Chapter 16). Owners act as gatekeepers for their particular directories. A change might be proposed by any engineer and LGTM'ed by any other engineer, but an owner of the directory in question must also *approve* this addition to their part of the codebase. Such an owner might be a tech lead or other engineer deemed expert in that particular area of the codebase. It's generally up to each team to decide how broadly or narrowly to assign ownership privileges.
- Approval from someone with language "readability"[3] that the code conforms to the language's style and best practices, checking whether the code is written in the manner we expect. This approval, again, might be implicit if the author has such readability. These engineers are pulled from a company-wide pool of engineers who have been granted readability in that programming language.

對於 Google 的任何特定變化，有三個方面的審查需要 "批准"：

- 由另一位工程師進行的正確性和可讀性檢查，檢查程式碼是否合適，以及程式碼是否符合作者的要求。這通常是一個團隊成員，儘管不一定是。這反映在 LGTM 許可權的 "標識 "上，在同行審查者同意程式碼 "看起來不錯 "之後，該標識將被設定。
- 來自程式碼所有者之一的批准，即該程式碼適合於程式碼庫的這個特定部分（並且可以被檢查到一個特定的目錄）。如果作者是這樣一個所有者，這種批准可能是隱含的。谷歌的程式碼函式庫是一個樹狀結構，有特定目錄的分層所有者。(見第 16 章）。所有者作為他們特定目錄的看門人。任何工程師都可以提出修改意見，任何其他工程師也可以提出 LGTM，但有關目錄的所有者*必須批准*在他們的程式碼庫中加入這一內容。這樣的所有者可能是技術領導或其他被認為是程式碼庫特定領域的專家的工程師。通常由每個團隊決定分配所有權特權的範圍是寬還是窄。
- 擁有語言 "可讀性"的人批准程式碼符合語言的風格和最佳實踐，檢查程式碼是否以我們期望的方式編寫。如果作者有這樣的可讀性，這種認可又可能是隱含的。這些工程師來自公司範圍內的工程師隊伍，他們被授予了該程式語言的可讀性。

Although this level of control sounds onerous—and, admittedly, it sometimes is— most reviews have one person assuming all three roles, which speeds up the process quite a bit. Importantly, the author can also assume the latter two roles, needing only an LGTM from another engineer to check code into their own codebase, provided they already have readability in that language (which owners often do).

雖然這種程度的控制聽起來很繁瑣——而且，不可否認，有時的確如此——但大多數審查都是由一個人承擔這三個角色，這就大大加快了審查過程。重要的是，作者也可以承擔後兩個角色，只需要另一個工程師的 LGTM 就可以將程式碼檢查到他們自己的程式碼庫中，前提是他們已經具備了該語言的可讀性（所有者經常這樣做）。

These requirements allow the code review process to be quite flexible. A tech lead who is an owner of a project and has that code's language readability can submit a code change with only an LGTM from another engineer. An intern without such authority can submit the same change to the same codebase, provided they get approval from an owner with language readability. The three aforementioned permission "bits" can be combined in any combination. An author can even request more than one LGTM from separate people by explicitly tagging the change as wanting an LGTM from all reviewers.

這些要求使得程式碼審查過程非常靈活。技術負責人是一個專案的所有者，並且具有程式碼的語言可讀性，可以只使用另一個工程師的 LGTM 提交程式碼更改。沒有這種許可權的實習生可以將相同的更改提交給相同的程式碼庫，前提是他們獲得具有語言可讀性的所有者的批准。上述三個許可"標識"可以任意組合。作者甚至可以從不同的人處請求多個 LGTM，方法是明確地將更改標記為希望從所有審閱者處獲得 LGTM。

In practice, most code reviews that require more than one approval usually go through a two-step process: gaining an LGTM from a peer engineer, and then seeking approval from appropriate code owner/readability reviewer(s). This allows the two roles to focus on different aspects of the code review and saves review time. The primary reviewer can focus on code correctness and the general validity of the code change; the code owner can focus on whether this change is appropriate for their part of the codebase without having to focus on the details of each line of code. An approver is often looking for something different than a peer reviewer, in other words. After all, someone is trying to check in code to their project/directory. They are more concerned with questions such as: "Will this code be easy or difficult to maintain?" "Does it add to my technical debt?" "Do we have the expertise to maintain it within our team?"

在實踐中，大多數需要不止一次批准的程式碼評審通常經歷兩個步驟：從同行工程師那裡獲得 LGTM，然後從適當的程式碼所有者/可讀性審查員處尋求批准。這使得這兩個角色可以專注於程式碼審查的不同方面，並節省審查時間。主要的審查者可以專注於程式碼的正確性和程式碼修改的一般有效性；程式碼所有者可以關注此更改是否適合他們的部分，而不必關注每行程式碼的細節。換句話說，審批者所尋找的東西往往與同行評審者不同。畢竟，有人是想把程式碼簽入他們的專案/目錄。他們更關心的是諸如以下問題。"這段程式碼是容易還是難以維護？" "它是否增加了我的技術債務？" "我們的團隊中是否有維護它的專業知識？"

If all three of these types of reviews can be handled by one reviewer, why not just have those types of reviewers handle all code reviews? The short answer is scale. Separating the three roles adds flexibility to the code review process. If you are working with a peer on a new function within a utility library, you can get someone on your team to review the code for code correctness and comprehension. After several rounds (perhaps over several days), your code satisfies your peer reviewer and you get an LGTM. Now, you need only get an *owner* of the library (and owners often have appropriate readability) to approve the change.

如果這三種類型的審查都可以由一個審查員處理，為什麼不直接讓這些型別的審查員處理所有的程式碼審查呢？簡單的答案是規模。將這三種角色分開，可以增加程式碼審查過程的靈活性。如果你和同行一起在一個實用程式庫中開發一個新的函式，你可以讓你團隊中的某人來審查程式碼的正確性和理解性。經過幾輪（也許是幾天的時間），你的程式碼讓你的同行審查員滿意，你就會得到一個 LGTM。現在，你只需要讓該庫的*所有者*（而所有者往往具有適當的可讀性）批准這項修改。

> 3 在谷歌，“可讀性”不僅僅指理解能力，而是指允許其他工程師維護程式碼的一套風格和最佳實踐。見第 3 章。

Ownership 所有權

Hyrum Wright

When working on a small team in a dedicated repository, it's common to grant the entire team access to everything in the repository. After all, you know the other engineers, the domain is narrow enough that each of you can be experts, and small numbers constrain the effect of potential errors.

當一個小團隊在專門的版本庫中工作時，通常會授予整個團隊對版本庫中所有內容的存取權。畢竟，你認識其他的工程師，這個領域很窄，你們每個人都可以成為專家，而且人數少限制了潛在錯誤的影響範圍。

As the team grows larger, this approach can fail to scale. The result is either a messy repository split or a different approach to recording who has what knowledge and responsibilities in different parts of the repository. At Google, we call this set of knowledge and responsibilities *ownership* and the people to exercise them *owners*. This concept is different than possession of a collection of source code, but rather implies a sense of stewardship to act in the company's best interest with a section of the codebase. (Indeed, "stewards" would almost certainly be a better term if we had it to do over again.)

隨著團隊規模的擴大，這種方法可能無法擴充。其結果要麼是混亂的版本庫分裂，要麼是用不同的方法來記錄誰在版本庫的不同部分擁有哪些知識和職責。在谷歌，我們把這套知識和職責稱為*所有權*，把行使這些知識和職責的人稱為*所有者*。這個概念不同於對原始碼集合的佔有，而是意味著一種管理意識，以公司的最佳利益對程式碼庫的某個部分採取行動。(事實上，如果我們重新來過，"管家 "幾乎肯定是一個更好的術語）。

Specially named OWNERS files list usernames of people who have ownership responsibilities for a directory and its children. These files may also contain references to other OWNERS files or external access control lists, but eventually they resolve to a list of individuals. Each subdirectory may also contain a separate OWNERS file, and the relationship is hierarchically additive: a given file is generally owned by the union of the members of all the OWNERS files above it in the directory tree. OWNERS files may have as many entries as teams like, but we encourage a relatively small and focused list to ensure responsibility is clear.

特別命名的 OWNERS 檔案列出了對一個目錄及其子目錄有所有權責任的人的使用者名稱。這些檔案也可能包含對其他 OWNERS 檔案或外部存取控制列表的參考，但最終它們會解析為個人列表。每個子目錄也可能包含一個單獨的 OWNERS 檔案，而且這種關係是分層遞增的：一個給定的檔案通常由目錄樹中它上面的所有 OWNERS 檔案的成員共同擁有。OWNERS 檔案可以有任意多的條目，但我們鼓勵一個相對較小和集中的列表，以確保責任明確。

Ownership of Google's code conveys approval rights for code within one's purview, but these rights also come with a set of responsibilities, such as understanding the code that is owned or knowing how to find somebody who does. Different teams have different criteria for granting ownership to new members, but we generally encourage them not to use ownership as a rite of initiation and encourage departing members to yield ownership as soon as is practical.

對谷歌程式碼的所有權傳達了對其許可權範圍內的程式碼的批准權，但這些權利也伴隨著一系列的責任，如瞭解所擁有的程式碼或知道如何找到擁有這些程式碼的人。不同的團隊在授予新成員所有權方面有不同的標準，但我們通常鼓勵他們不要把所有權作為入職儀式，並鼓勵離職的成員在可行的情況下儘快放棄所有權。

This distributed ownership structure enables many of the other practices we've outlined in this book. For example, the set of people in the root OWNERS file can act as global approvers for large-scale changes (see Chapter 22) without having to bother local teams. Likewise, OWNERS files act as a kind of documentation, making it easy for people and tools to find those responsible for a given piece of code just by walking up the directory tree. When new projects are created, there's no central authority that has to register new ownership privileges: a new OWNERS file is sufficient.

這種分散式所有權結構使我們在本書中概述的許多其他做法成為可能。例如，根 OWNERS 檔案中的一組人可以作為大規模修改的全球批准者（見第 22 章），而不必打擾本地團隊。同樣，OWNERS 檔案作為一種文件，使人們和工具很容易找到對某段程式碼負責的人，只要沿著目錄樹走就可以了。當新專案被建立時，沒有一箇中央機構需要註冊新的所有權許可權：一個新的 OWNERS 檔案就足夠了。

This ownership mechanism is simple, yet powerful, and has scaled well over the past two decades. It is one of the ways that Google ensures that tens of thousands of engineers can operate efficiently on billions of lines of code in a single repository.

這種所有權機制簡單而強大，在過去的 20 年裡得到了良好的擴充。這是谷歌確保數以萬計的工程師能夠在單一資源庫中有效運算元十億行程式碼的方法之一。

Code Review Benefits 程式碼審查的好處

Across the industry, code review itself is not controversial, although it is far from a universal practice. Many (maybe even most) other companies and open source projects have some form of code review, and most view the process as important as a sanity check on the introduction of new code into a codebase. Software engineers understand some of the more obvious benefits of code review, even if they might not personally think it applies in all cases. But at Google, this process is generally more thorough and wide spread than at most other companies.

縱觀整個行業，程式碼審查本身並不存在爭議，儘管它還遠不是一種普遍的做法。許多（甚至可能是大多數）其他公司和開源專案都有某種形式的程式碼審查，而且大多數人認為這個過程很重要，是對引入新程式碼到程式碼庫的合理檢查。軟體工程師理解程式碼審查的一些更明顯的好處，即使他們個人可能不認為它適用於所有情況。但在谷歌，這個過程通常比其他大多數公司更徹底、更廣泛。

Google's culture, like that of a lot of software companies, is based on giving engineers wide latitude in how they do their jobs. There is a recognition that strict processes tend not to work well for a dynamic company needing to respond quickly to new technologies, and that bureaucratic rules tend not to work well with creative professionals. Code review, however, is a mandate, one of the few blanket processes in which all software engineers at Google must participate. Google requires code review for almost [4] every code change to the codebase, no matter how small. This mandate does have a cost and effect on engineering velocity given that it does slow down the introduction of new code into a codebase and can impact time-to-production for any given code change. (Both of these are common complaints by software engineers of strict code review processes.) Why, then, do we require this process? Why do we believe that this is a long-term benefit?

谷歌的文化，就像許多軟體公司的文化一樣，是基於給工程師們在工作中的自由度。人們認識到，對於需要對新技術做出快速反應的充滿活力的公司來說，嚴格的流程往往不起作用，而官僚主義的規則往往不適合創造性專業人士。然而，程式碼審查是一項任務，是谷歌所有軟體工程師都必須參與的少數全流程之一。谷歌要求對程式碼庫的每一次程式碼修改都要進行程式碼審查，無論多麼微小。這個任務確實對工程速度有成本和影響，因為它確實減緩了將新程式碼引入程式碼庫的速度，並可能影響任何特定程式碼更改的生產時間。(這兩點是軟體工程師對嚴格的程式碼審查過程的常見抱怨）。那麼，為什麼我們要要求這個過程？為什麼我們相信這是一個長期有利的？

A well-designed code review process and a culture of taking code review seriously provides the following benefits:

- Checks code correctness
- Ensures the code change is comprehensible to other engineers
- Enforces consistency across the codebase
- Psychologically promotes team ownership
- Enables knowledge sharing
- Provides a historical record of the code review itself

一個精心設計的程式碼審查過程和認真對待程式碼審查的文化會帶來以下好處：

- 檢查程式碼的正確性
- 確保其他工程師能夠理解程式碼更改

- 強化整個程式碼庫的一致性
- 從心理上促進團隊的所有權
- 實現知識共享
- 提供程式碼審查本身的歷史記錄

Many of these benefits are critical to a software organization over time, and many of them are beneficial to not only the author but also the reviewers. The following sections go into more specifics for each of these items.

隨著時間的推移，這些好處對一個軟體組織來說是至關重要的，其中許多好處不僅對作者有利，而且對審查員也有利。下面的章節將對這些專案中的每一項進行更詳細的說明。

> 4 對文件和配置的某些更改可能不需要程式碼審查，但通常仍然可以獲得這樣的審查。

Code Correctness 程式碼正確性

An obvious benefit of code review is that it allows a reviewer to check the “correctness” of the code change. Having another set of eyes look over a change helps ensure that the change does what was intended. Reviewers typically look for whether a change has proper testing, is properly designed, and functions correctly and efficiently. In many cases, checking code correctness is checking whether the particular change can introduce bugs into the codebase.

程式碼審查的一個明顯的好處是，它允許審查者檢查程式碼更改的 "正確性"。讓另一雙眼睛來審視一個更改，有助於確保這個更改能達到預期效果。審查員通常會檢查一個變更是否有適當的測試，設計是否合理，功能是否正確和有效。在許多情況下，檢查程式碼正確性就是檢查特定的更改是否會將 bug 引入程式碼庫。

Many reports point to the efficacy of code review in the prevention of future bugs in software. A study at IBM found that discovering defects earlier in a process, unsurprisingly, led to less time required to fix them later on.[5] The investment in the time for code review saved time otherwise spent in testing, debugging, and performing regressions, provided that the code review process itself was streamlined to keep it lightweight. This latter point is important; code review processes that are heavyweight, or that don’t scale properly, become unsustainable.[6] We will get into some best practices for keeping the process lightweight later in this chapter.

許多報告指出了程式碼審查在防止軟體未來出現錯誤方面的有效性。IBM 的一項研究發現，在一個過程的越早發現缺陷，無疑會減少以後修復缺陷所需的時間。對程式碼審查時間的投入節省了原本用於測試、除錯和執行迴歸的時間，前提是程式碼審查過程本身經過了最佳化，以保持其輕量級。如果程式碼審查過程很重，或者擴充不當，那麼這些過程將變得不可持續。我們將在本章後面介紹一些保持過程輕量級的最佳實踐。

To prevent the evaluation of correctness from becoming more subjective than objective, authors are generally given deference to their particular approach, whether it be in the design or the function of the introduced change. A reviewer shouldn’t propose alternatives because of personal opinion. Reviewers can propose alternatives, but only if they improve comprehension (by being less complex, for example) or functionality (by being more efficient, for example). In general, engineers are encouraged to approve changes that improve the codebase rather than wait for consensus on a more “perfect” solution. This focus tends to

speed up code reviews.

為了防止正確性評估變得更加主觀而非客觀，作者通常會遵循其特定方法，無論是在設計中還是在引入變更的功能中。審查員不應該因為個人意見而提出替代方案。審查員可以提出替代方案，但前提是這些替代方案能夠改善理解性（例如，透過降低複雜性）或功能性（例如，透過提高效率）。一般來說，鼓勵工程師批准改進程式碼庫的更改，而不是等待就更“完美”的解決方案達成共識。這種關注傾向於加速程式碼審查。

As tooling becomes stronger, many correctness checks are performed automatically through techniques such as static analysis and automated testing (though tooling might never completely obviate the value for human-based inspection of code—see Chapter 20 for more information). Though this tooling has its limits, it has definitely lessoned the need to rely on human-based code reviews for checking code correctness.

隨著工具越來越強大，許多正確性檢查會透過靜態分析和自動測試等技術自動執行（儘管工具可能永遠不會完全消除基於人工的程式碼檢查的價值，更多資訊請參見第 20 章）。儘管這種工具有其侷限性，但它明確地說明了需要依靠基於人工的程式碼檢查來檢查程式碼的正確性。

That said, checking for defects during the initial code review process is still an integral part of a general “shift left” strategy, aiming to discover and resolve issues at the earliest possible time so that they don’t require escalated costs and resources farther down in the development cycle. A code review is neither a panacea nor the only check for such correctness, but it is an element of a defense-in-depth against such problems in software. As a result, code review does not need to be “perfect” to achieve results.

這就是說，在最初的程式碼審查過程中檢查缺陷仍然是一般“左移”策略的一個組成部分，旨在儘早發現和解決問題，從而避免在開發週期中進一步增加成本和資源。程式碼審查既不是萬靈藥，也不是檢查這種正確性的唯一方法，但它是深入防禦軟體中此類問題的一個要素。因此，程式碼審查不需要“完美”才能取得成果。

Surprisingly enough, checking for code correctness is not the primary benefit Google accrues from the process of code review. Checking for code correctness generally ensures that a change works, but more importance is attached to ensuring that a code change is understandable and makes sense over time and as the codebase itself scales. To evaluate those aspects, we need to look at factors other than whether the code is simply logically “correct” or understood.

令人驚訝的是，檢查程式碼的正確性並不是谷歌從程式碼審查過程中獲得的最大好處。檢查程式碼正確性通常可以確保更改有效，但更重要的是確保程式碼更改是可以理解的，並且隨著時間的推移和程式碼庫本身的擴充而變得有意義。為了評估這些方面，我們需要檢視除程式碼在邏輯上是否“正確”或理解之外的其他因素。

> 5 "Advances in Software Inspection," IEEE Transactions on Software Engineering, SE-12(7): 744-751, July 1986. 誠然，這項研究發生在強大的工具和自動測試在軟體開發過程中變得如此重要之前，但其結果在現代軟體時代似乎仍有意義。
>
> 6 Rigby, Peter C. and Christian Bird. 2013. "趨同的軟體同行評審實踐"。ESEC/FSE 2013。2013 年第九屆軟體工程基礎聯席會議論文集》，2013 年 8 月：202-212。https:// dl.acm.org/doi/10.1145/2491411.2491444。

Comprehension of Code 程式碼理解

A code review typically is the first opportunity for someone other than the author to inspect a change. This perspective allows a reviewer to do something that even the best engineer cannot do: provide feedback unbiased by an author's perspective. *A code review is often the first test of whether a given change is understandable to a broader audience*. This perspective is vitally important because code will be read many more times than it is written, and understanding and comprehension are critically important.

程式碼審查通常是作者以外的人檢查修改的第一個時機。這種視角使審查者能夠做到最好的工程師也做不到的事情：提供不受作者視角影響的反饋。*程式碼審查通常是對一個特定的變更是否能被更多的人理解的第一個測試*。這種觀點是非常重要的，因為程式碼被閱讀的次數要比它被寫的次數多得多，而理解和領悟是非常重要的。

It is often useful to find a reviewer who has a different perspective from the author, especially a reviewer who might need, as part of their job, to maintain or use the code being proposed within the change. Unlike the deference reviewers should give authors regarding design decisions, it's often useful to treat questions on code comprehension using the maxim "the customer is always right." In some respect, any questions you get now will be multiplied many-fold over time, so view each question on code comprehension as valid. This doesn't mean that you need to change your approach or your logic in response to the criticism, but it does mean that you might need to explain it more clearly.

找到一個與作者觀點不同的讀者通常是很有用的，特別是一個審查員，作為他們工作的一部分，可能需要維護或使用修改中提出的程式碼。與審查員在設計決策方面應該給予作者的尊重不同，用 "客戶永遠是對的 "這一格言來對待程式碼理解方面的問題往往是有用的。在某種程度上，你現在得到的任何問題都會隨著時間的推移而成倍增加，所以要把每個關於程式碼理解的問題看作是有效的。這並不意味著你需要改變你的方法或邏輯來回應批評，但這確實意味著你可能需要更清楚地解釋它。

Together, the code correctness and code comprehension checks are the main criteria for an LGTM from another engineer, which is one of the approval bits needed for an approved code review. When an engineer marks a code review as LGTM, they are saying that the code does what it says and that it is understandable. Google, however, also requires that the code be sustainably maintained, so we have additional approvals needed for code in certain cases.

程式碼正確性和程式碼理解力的檢查共同構成了另一個工程師的 LGTM 的主要標準，這也是一個被批准的程式碼審查所需的批准之一。當一個工程師將程式碼審查標記為 LGTM 時，他們是在說，程式碼做了它所說的事情，而且它是可以理解的。然而，谷歌也要求程式碼是可持續維護的，所以我們在某些情況下對程式碼還需要額外的批准。

Code Consistency 程式碼的一致性

At scale, code that you write will be depended on, and eventually maintained, by someone else. Many others will need to read your code and understand what you did. Others (including automated tools) might need to refactor your code long after you've moved to another project. Code, therefore, needs to conform to some standards of consistency so that it can be understood and maintained. Code should also avoid being overly complex; simpler code is easier for others to understand and maintain as well. Reviewers can assess how well this code lives up to the standards of the codebase itself during code review. A code review, therefore, should act to ensure *code health*.

在規模上，你寫的程式碼會被別人依賴，並最終由其他人維護。許多人需要閱讀你的程式碼並瞭解你的工作。在你轉移到另一個專案之後很長時間後，其他人（包括自動化工具）可能需要重構你的程式碼。因此，程式碼需要符合一些一致性的標準，這樣它才能被理解和維護。程式碼也應避免過於複雜；簡單的程式碼對其他人來說也更容易理解和維護。審查員可以在程式碼評審中評估這些程式碼是否符合程式碼庫本身的標準。因此，程式碼審查的作用應該是確保*程式碼的健康*。

It is for maintainability that the LGTM state of a code review (indicating code correctness and comprehension) is separated from that of readability approval. Readability approvals can be granted only by individuals who have successfully gone through the process of code readability training in a particular programming language. For example, Java code requires approval from an engineer who has “Java readability.”

正是為了可維護性，程式碼審查的 LGTM 狀態（表示程式碼的正確性和理解力）與可讀性批准的狀態是分開的。可讀性的批准只能由成功透過特定程式語言的程式碼可讀性培訓過程的人授予。例如，Java 程式碼需要有 "Java 可讀性 "的工程師來批准。

A readability approver is tasked with reviewing code to ensure that it follows agreedon best practices for that particular programming language, is consistent with the codebase for that language within Google’s code repository, and avoids being overly complex. Code that is consistent and simple is easier to understand and easier for tools to update when it comes time for refactoring, making it more resilient. If a particular pattern is always done in one fashion in the codebase, it’s easier to write a tool to refactor it.

可讀性審查員的任務是審查程式碼，以確保它遵循該特定程式語言的商定的最佳做法，與谷歌程式碼庫中該語言的程式碼庫一致，並避免過於複雜。一致和簡單的程式碼更容易理解，當需要重構時，工具也更容易更新，使其更有擴充性。如果一個特定的模式在程式碼庫中總是以一種方式完成，那麼寫一個工具來重構它就會更容易。

Additionally, code might be written only once, but it will be read dozens, hundreds, or even thousands of times. Having code that is consistent across the codebase improves comprehension for all of engineering, and this consistency even affects the process of code review itself. Consistency sometimes clashes with functionality; a readability reviewer may prefer a less complex change that may not be functionally “better” but is easier to understand.

此外，程式碼可能只編寫一次，但它將被讀取數十次、數百次甚至數千次。在整個程式碼庫中使用一致的程式碼可以提高對所有工程的理解，這種一致性甚至會影響程式碼評審本身的過程。一致性有時與功能衝突；可讀性審查員可能更喜歡不太複雜的更改，這些更改在功能上可能不是“更好”，但更容易理解。

With a more consistent codebase, it is easier for engineers to step in and review code on someone else’s projects. Engineers might occasionally need to look outside the team for help in a code review. Being able to reach out and ask experts to review the code, knowing they can expect the code itself to be consistent, allows those engineers to focus more properly on code correctness and comprehension.

有了一個更加一致的程式碼庫，工程師就更容易介入並審查別人專案的程式碼。工程師們可能偶爾需要向團隊外尋求程式碼審查方面的幫助。能夠伸出援手，請專家來審查程式碼，知道他們可以期望程式碼本身是一致的，使這些工程師能夠更正確地關注程式碼的正確性和理解。

Psychological and Cultural Benefits 心理和文化方面的好處

Code review also has important cultural benefits: it reinforces to software engineers that code is not “theirs” but in fact part of a collective enterprise. Such psychological benefits can be subtle but are still important. Without code review, most engineers would naturally gravitate toward personal style and their own approach to software design. The code review process forces an author to not only let others have input, but to compromise for the sake of the greater good.

程式碼審查還有重要的文化好處：它向軟體工程師強調程式碼不是“他們的”，而是事實上集體事業的一部分。這種心理上的好處可能很微妙，但仍然很重要。沒有程式碼審查，大多數工程師自然會傾向於個人風格和他們自己的軟體設計方法。程式碼審查過程迫使作者不僅要讓別人提出意見，而且要為了更大的利益做出妥協。

It is human nature to be proud of one’s craft and to be reluctant to open up one’s code to criticism by others. It is also natural to be somewhat reticent to welcome critical feedback about code that one writes. The code review process provides a mechanism to mitigate what might otherwise be an emotionally charged interaction. Code review, when it works best, provides not only a challenge to an engineer’s assumptions, but also does so in a prescribed, neutral manner, acting to temper any criticism which might otherwise be directed to the author if provided in an unsolicited manner. After all, the process *requires* critical review (we in fact call our code review tool “Critique”), so you can’t fault a reviewer for doing their job and being critical. The code review process itself, therefore, can act as the “bad cop,” whereas the reviewer can still be seen as the “good cop.”

以自己的手藝為榮，不願意公開自己的程式碼接受他人的批評，這是人類別的天性。對於自己寫的程式碼的批評性反饋，也很自然地不願意接受。程式碼審查過程提供了一個機制，以減輕可能是一個情緒化的互動。如果程式碼審查效果最佳，它不僅會對工程師的假設提出質疑，而且還會以規定的、中立的方式提出質疑，如果以未經請求的方式提出批評，則可能會對作者提出批評。畢竟，這個過程需要批判性的審查（事實上，我們把我們的程式碼審查工具稱為 "批判"），所以你不能責怪審查者做他們的工作和批評。因此，程式碼審查過程本身可以充當 "壞警察"，而審查者仍然可以被看作是 "好警察"。

Of course, not all, or even most, engineers need such psychological devices. But buffering such criticism through the process of code review often provides a much gentler introduction for most engineers to the expectations of the team. Many engineers joining Google, or a new team, are intimidated by code review. It is easy to think that any form of critical review reflects negatively on a person’s job performance. But over time, almost all engineers come to expect to be challenged when sending a code review and come to value the advice and questions offered through this process (though, admittedly, this sometimes takes a while).

當然，不是所有的，甚至是大多數的工程師都需要這樣的心理措施。但是，透過程式碼審查的過程來緩解這種批評，往往能為大多數工程師提供一個更溫和的指導，讓他們瞭解團隊的期望。許多加入谷歌的工程師，或者一個新的團隊，都被程式碼審查所嚇倒。我們很容易認為任何形式的批評性審查都會對一個人的工作表現產生負面影響。但是，隨著時間的推移，幾乎所有的工程師都期望在傳送程式碼審查時受到挑戰，並開始重視透過這一過程提供的建議和問題（儘管，無可否認，這有時需要一段時間）。

Another psychological benefit of code review is validation. Even the most capable engineers can suffer from imposter syndrome and be too self-critical. A process like code review acts as validation and recognition for one’s work. Often, the process involves an exchange of ideas and knowledge sharing (covered in the next section), which benefits both the reviewer and the reviewee. As an engineer grows in their domain knowledge, it’s sometimes difficult for them to get positive feedback on how they improve. The process of code review can provide that mechanism.

程式碼審查的另一個心理好處是驗證。即使是最有能力的工程師也可能患上冒名頂替綜合症，過於自我批評。像程式碼評審這樣的過程是對一個人工作的確認和認可。通常，該過程涉及思想交流和知識共享（將在下一節中介紹），這對稽核人和被稽核人都有好處。隨著工程師領域知識的增長，他們有時很難獲得關於如何改進的積極反饋。程式碼審查過程可以提供這種機制。

The process of initiating a code review also forces all authors to take a little extra care with their changes. Many software engineers are not perfectionists; most will admit that code that “gets the job done” is better than code that is perfect but that takes too long to develop. Without code review, it’s natural that many of us would cut corners, even with the full intention of correcting such defects later. “Sure, I don’t have all of the unit tests done, but I can do that later.” A code review forces an engineer to resolve those issues before sending the change. Collecting the components of a change for code review psychologically forces an engineer to make sure that all of their ducks are in a row. The little moment of reflection that comes before sending off your change is the perfect time to read through your change and make sure you’re not missing anything.

啟動程式碼審查的過程也迫使所有作者對他們的更改多加註意。許多軟體工程師並不是完美主義者；大多數人都會承認，"能完成工作 "的程式碼要比完美但開發時間太長的程式碼要好。我們中的許多人會抄近路是很自然的，即使我們完全打算在以後糾正這些缺陷。"當然，我沒有完成所有的單元測試，但我可以以後再做。" 程式碼審查迫使工程師在傳送修改前解決這些問題。從心理上講，為程式碼審查而收集修改的組成部分，迫使工程師確保他們所有的事情都是一帆風順的。在傳送修改前的那一小段思考時間是閱讀修改的最佳時機，以確保你沒有遺漏任何東西。

Knowledge Sharing 知識共享

One of the most important, but underrated, benefits of code review is in knowledge sharing. Most authors pick reviewers who are experts, or at least knowledgeable, in the area under review. The review process allows reviewers to impart domain knowledge to the author, allowing the reviewer(s) to offer suggestions, new techniques, or advisory information to the author. (Reviewers can even mark some comments “FYI,” requiring no action; they are simply added as an aid to the author.) Authors who become particularly proficient in an area of the codebase will often become owners as well, who then in turn will be able to act as reviewers for other engineers.

程式碼審查的一個最重要的，但被低估的好處是知識共享。大多數作者挑選的審查員都是被審查領域的專家，或者至少在所審查的領域有知識。審查過程允許審查員向作者傳授領域知識，允許審查員向作者提供建議、新技術或諮詢資訊。(審查員甚至可以把一些評論標記為 "僅供參考"，不需要採取任何行動；它們只是作為作者的一種幫助而被新增進來）。在程式碼庫某個領域特別精通的作者通常也會成為所有者，而後者又可以作為其他工程師的評審員。

Part of the code review process of feedback and confirmation involves asking questions on why the change is done in a particular way. This exchange of information facilitates knowledge sharing. In fact, many code reviews involve an exchange of information both ways: the authors as well as the reviewers can learn new techniques and patterns from code review. At Google, reviewers may even directly share suggested edits with an author within the code review tool itself.

反饋和確認的程式碼評審過程的一部分包括詢問為什麼以特定方式進行更改。這種資訊交流有助於知識共享。事實上，許多程式碼評審都涉及雙向資訊交換：作者和審查員都可以從程式碼評審中學習新的技術和模式。在谷歌，審查員甚至可以直接在程式碼審查工具中與作者分享建議的編輯。

An engineer may not read every email sent to them, but they tend to respond to every code review sent. This knowledge sharing can occur across time zones and projects as well, using Google's scale to disseminate information quickly to engineers in all corners of the codebase. Code review is a perfect time for knowledge transfer: it is timely and actionable. (Many engineers at Google "meet" other engineers first through their code reviews!)

工程師可能不會閱讀每一封發給他們的電子郵件，但他們往往會回覆每一封傳送的程式碼審查。這種知識共享也可以跨越時區和專案，利用谷歌的規模將資訊迅速傳播到程式碼庫的各個角落的工程師。程式碼審查是知識轉移的最佳時機：它是及時和可操作的。(谷歌的許多工程師首先透過程式碼審查 "認識 "其他工程師！）。

Given the amount of time Google engineers spend in code review, the knowledge accrued is quite significant. A Google engineer's primary task is still programming, of course, but a large chunk of their time is still spent in code review. The code review process provides one of the primary ways that software engineers interact with one another and exchange information about coding techniques. Often, new patterns are advertised within the context of code review, sometimes through refactorings such as large-scale changes.

考慮到谷歌工程師花在程式碼審查上的時間，積累的知識相當重要。當然，谷歌工程師的主要任務仍然是程式設計，但他們的大部分時間仍然花在程式碼審查上。程式碼評審過程提供了軟體工程師相互交流和交換編碼技術資訊的主要方式之一。通常，新模式是在程式碼審查的上下文中釋出的，有時是透過重構（如大規模更改）釋出的。

Moreover, because each change becomes part of the codebase, code review acts as a historical record. Any engineer can inspect the Google codebase and determine when some particular pattern was introduced and bring up the actual code review in question. Often, that archeology provides insights to many more engineers than the original author and reviewer(s).

此外，由於每個更改都成為程式碼庫的一部分，所以程式碼評審充當歷史記錄。任何工程師都可以檢查谷歌的程式碼庫，並確定何時引入某些特定的模式，並提出有關的實際程式碼審查。通常情況下，這種考古學為比原作者和審查者更多的工程師提供了洞察力。

Code Review Best Practices 程式碼評審最佳實踐

Code review can, admittedly, introduce friction and delay to an organization. Most of these issues are not problems with code review per se, but with their chosen implementation of code review. Keeping the code review process running smoothly at Google is no different, and it requires a number of best practices to ensure that code review is worth the effort put into the process. Most of those practices emphasize keeping the process nimble and quick so that code review can scale properly.

誠然，程式碼審查會給一個組織帶來阻力和延遲。這些問題大多不是程式碼審查本身的問題，而是他們選擇的程式碼審查實施的問題。在谷歌保持程式碼審查過程的順利執行也不例外，它需要大量的最佳實踐來確保程式碼審查是值得的。大多數實踐都強調保持流程的敏捷性和快速性，以便程式碼評審能夠適當地擴充。

Be Polite and Professional 要有禮貌和專業精神

As pointed out in the Culture section of this book, Google heavily fosters a culture of trust and respect. This filters down into our perspective on code review. A software engineer needs an LGTM from only one other engineer to satisfy our requirement on code comprehension, for example. Many engineers make comments and LGTM a change with the understanding that the change can be submitted after those changes are made, without any additional rounds of review. That said, code reviews can introduce anxiety and stress to even the most capable engineers. It is critically important to keep all feedback and criticism firmly in the professional realm.

正如本書文化部分所指出的，谷歌大力培育信任和尊重的文化。這將深入到我們對程式碼審查的觀點中。例如，軟體工程師只需要一個來自其他工程師的 LGTM 就可以滿足我們對程式碼理解的要求。許多工程師在作出更改後提出意見和 LGTM，並理解這些更改可以在做出更改後提交，而無需進行任何額外的審查。也就是說，即使是最有能力的工程師，程式碼審查也會帶來焦慮和壓力。在專業領域中，堅定地保留所有反饋和批評是至關重要的。

In general, reviewers should defer to authors on particular approaches and only point out alternatives if the author's approach is deficient. If an author can demonstrate that several approaches are equally valid, the reviewer should accept the preference of the author. Even in those cases, if defects are found in an approach, consider the review a learning opportunity (for both sides!). All comments should remain strictly professional. Reviewers should be careful about jumping to conclusions based on a code author's particular approach. It's better to ask questions on why something was done the way it was before assuming that approach is wrong.

一般來說，審查員應該在特定的方法上聽從作者的意見，只有在作者的方法有缺陷時才指出替代方法。如果作者能證明幾種方法同樣有效，審查員應該接受作者的偏好。即使在這些情況下，如果發現一個方法有缺陷，也要把審檢視作是一個學習的機會（對雙方都是如此！）。所有的評論都應該嚴格保持專業性。審查員應該注意不要根據程式碼作者的特定方法就下結論。在假設該方法是錯誤的之前，最好先問一下為什麼要這樣做。

Reviewers should be prompt with their feedback. At Google, we expect feedback from a code review within 24 (working) hours. If a reviewer is unable to complete a review in that time, it's good practice (and expected) to respond that they've at least seen the change and will get to the review as soon as possible. Reviewers should avoid responding to the code review in piecemeal fashion. Few things annoy an author more than getting feedback from a review, addressing it, and then continuing to get unrelated further feedback in the review process.

審查員應及時提供反饋。在谷歌，我們希望在 24（工作）小時內得到程式碼審查的反饋。如果審查員無法在這段時間內完成審查，那麼良好的做法是（也是我們所期望的）回應說他們至少已經看到了更改，並將儘快進行審查。審查員應該避免以零散的方式回應程式碼評審。沒有什麼事情比從審查中得到反饋，解決它，然後繼續在審查過程中得到無關的進一步反饋更讓作者惱火。

As much as we expect professionalism on the part of the reviewer, we expect professionalism on the part of the author as well. Remember that you are not your code, and that this change you propose is not "yours" but the team's. After you check that piece of code into the codebase, it is no longer yours in any case. Be receptive to questions on your approach, and be prepared to explain why you did things in certain ways. Remember that part of the responsibility of an author is to make sure this code is understandable and maintainable for the future.

就像我們期望審查員有專業精神一樣，我們也期望作者有專業精神。記住，你不是你的程式碼，你提出的這個修改不是 "你的"，而是團隊的。在你把這段程式碼檢查到程式碼庫中後，它無論如何都不再是你的了。要樂於接受關於你的方法的問題，並準備好解釋你為什麼以某種方式做事情。記住，作者的部分責任是確保這段程式碼是可以理解的，並且可以為將來維護。

It's important to treat each reviewer comment within a code review as a TODO item; a particular comment might not need to be accepted without question, but it should at least be addressed. If you disagree with a reviewer's comment, let them know, and let them know why and don't mark a comment as resolved until each side has had a chance to offer alternatives. One common way to keep such debates civil if an author doesn't agree with a reviewer is to offer an alternative and ask the reviewer to PTAL (please take another look). Remember that code review is a learning opportunity for both the reviewer and the author. That insight often helps to mitigate any chances for disagreement.

重要的是要把程式碼審查中的每個審查者的評論當作一個 TODO 專案；一個特定的評論可能不需要被無條件接受，但它至少應該被解決。如果你不同意一個評審員的評論，讓他們知道，並讓他們知道為什麼，在雙方都有機會提供替代方案之前，不要把評論標記為已解決。如果作者不同意審查員的意見，保持這種辯論的一個常見方法是提供一個替代方案，並要求評審員 PTAL（請再看看）。記住，程式碼審查對於審查者和作者來說都是一個學習的機會。這種洞察力往往有助於減少任何分歧的場景。

By the same token, if you are an owner of code and responding to a code review within your codebase, be amenable to changes from an outside author. As long as the change is an improvement to the codebase, you should still give deference to the author that the change indicates something that could and should be improved.

同樣的道理，如果你是程式碼的所有者，並且在你的程式碼庫中對程式碼審查做出回應，那麼就應該對來自外部作者的改動持寬容態度。只要這個改動是對程式碼庫的改進，你就應該尊重作者的意見，即更改表明了一些可以而且應該改進的地方。

Write Small Changes 寫出小的更改

Probably the most important practice to keep the code review process nimble is to keep changes small. A code review should ideally be easy to digest and focus on a single issue, both for the reviewer and the author. Google's code review process discourages massive changes consisting of fully formed projects, and reviewers can rightfully reject such changes as being too large for a single review. Smaller changes also prevent engineers from wasting time waiting for reviews on larger changes, reducing downtime. These small changes have benefits further down in the software development process as well. It is far easier to determine the source of a bug within a change if that particular change is small enough to narrow it down.

要保持程式碼審查過程的靈活性，最重要的做法可能是保持小的更改。理想情況下，程式碼審查應該是容易理解的，並且對審查者和作者來說，都是集中在一個問題上。谷歌的程式碼審查過程不鼓勵由完全成型的專案組成的大規模修改，審查員可以理所當然地拒絕這樣的更改，因為它對於一次審查來說太大。較小的改動也可以防止工程師浪費時間等待對較大變更的審查，減少停滯時間。這些小更改在軟體開發過程中也有好處。如果一個特定的變更足夠小，那麼確定該變更中的錯誤來源就容易得多。

That said, it's important to acknowledge that a code review process that relies on small changes is sometimes difficult to reconcile with the introduction of major new features. A set of small, incremental code changes can be easier to digest individually, but more difficult to comprehend within a larger scheme. Some engineers at Google admittedly are not fans of the preference given to small changes. Techniques exist for managing such code changes (development on integration branches, management of changes using a diff base different than HEAD), but those techniques inevitably involve more overhead. Consider the optimization for small changes just that: an optimization, and allow your process to accommodate the occasional larger change.

儘管如此，必須承認，依賴於小更改的程式碼審查過程有時很難與主要新特性的引入相協調。一組小的、漸進式的程式碼修改可能更容易單獨消化，但在一個更大的方案中卻更難理解。不可否認，谷歌的一些工程師並不喜歡小改動。存在管理這種程式碼變化的技術（在整合分支上開發，使用不同於 HEAD 的 diff base 管理變化），但這些技術不可避免地涉及更多的開銷。考慮到對小改動的最佳化只是一個最佳化，並允許你的過程適應偶爾的大更改。

Small" changes should generally be limited to about 200 lines of code. A small change should be easy on a reviewer and, almost as important, not be so cumbersome that additional changes are delayed waiting for an extensive review. Most changes at Google are expected to be reviewed within about a day.[7] (This doesn't necessarily mean that the review is over within a day, but that initial feedback is provided within a day.) About 35% of the changes at Google are to a single file.[8] Being easy on a reviewer allows for quicker changes to the codebase and benefits the author as well. The author wants a quick review; waiting on an extensive review for a week or so would likely impact follow-on changes. A small initial review also can prevent much more expensive wasted effort on an incorrect approach further down the line.

"小"改動一般應限制在 200 行左右的程式碼。一個小的更改應該對審查者來說是容易的，而且，幾乎同樣重要的是，不要太麻煩，以至於更多的更改被延遲，以等待廣泛的審查。在谷歌，大多數的更改預計會在一天內被審查。(這並不一定意味著審查在一天內結束，但初步反饋會在一天內提供。) 在谷歌，大約 35%的修改是針對單個檔案的。對審查者來說容易，可以更快地修改程式碼庫，對作者也有利。作者希望快速審查；等待一個星期左右的廣泛審查可能會影響後續的更改。一個小規模的初步審查也可以防止在後續的錯誤方法上浪費更昂貴的精力。

Because code reviews are typically small, it's common for almost all code reviews at Google to be reviewed by one and only one person. Were that not the case—if a team were expected to weigh in on all changes to a common codebase—there is no way the process itself would scale. By keeping the code reviews small, we enable this optimization. It's not uncommon for multiple people to comment on any given change— most code reviews are sent to a team member, but also CC'd to appropriate teams— but the primary reviewer is still the one whose LGTM is desired, and only one LGTM is necessary for any given change. Any other comments, though important, are still optional.

因為程式碼審查通常是小規模的，所以在谷歌，幾乎所有的程式碼審查都是由一個人審查，而且只有一個人。如果不是這樣的話——如果一個團隊被期望對一個共同的程式碼庫的所有更改進行評估，那麼這個過程本身就沒有辦法擴充。透過保持小規模的程式碼審查，我們實現了這種最佳化。多人對任何給定的更改發表評論的情況並不罕見--大多數程式碼審查被髮送給一個團隊成員，但也被抄送給適當的團隊——但主要的審查員仍然是那個希望得到 LGTM 的人，而且對於任何給定的更改，只有一個 LGTM 是必要的。任何其他評論，儘管很重要，但仍然是可選的。

Keeping changes small also allows the "approval" reviewers to more quickly approve any given changes. They can quickly inspect whether the primary code reviewer did due diligence and focus purely on whether this change augments the codebase while maintaining code health over time.

保持小的更改也允許 "批准 "審查員更快地批准任何特定的變化。他們可以快速檢查主要的程式碼審查員是否盡職盡責，並純粹關注這一變化是否增強了程式碼庫，同時隨著時間的推移保持程式碼的健康。

> 7 Caitlin Sadowski、Emma Söderberg、Luke Church、Michal Sipko 和 Alberto Baccelli，"現代程式碼評論：谷歌的案例研究"
>
> 8 同上。

Write Good Change Descriptions 寫出好的變更描述

A change description should indicate its type of change on the first line, as a summary. The first line is prime real estate and is used to provide summaries within the code review tool itself, to act as the subject line in any associated emails, and to become the visible line Google engineers see in a history summary within Code Search (see Chapter 17), so that first line is important.

變更描述應該在第一行標註它的變更型別，作為一個摘要。第一行是最重要的，它被用來在程式碼審查工具中提供摘要，作為任何相關電子郵件的主題行，併成為谷歌工程師在程式碼搜尋中看到的歷史摘要的可見行（見第 17 章），所以第一行很重要。

Although the first line should be a summary of the entire change, the description should still go into detail on what is being changed *and why*. A description of "Bug fix" is not helpful to a reviewer or a future code archeologist. If several related modifications were made in the change, enumerate them within a list (while still keeping it on message and small). The description is the historical record for this change, and tools such as Code Search allow you to find who wrote what line in any particular change in the codebase. Drilling down into the original change is often useful when trying to fix a bug.

雖然第一行應該是整個更改的摘要，但描述仍然應該詳細說明更改的內容和*原因*。"Bug 修復"的描述對審查員或未來的程式碼考古學家來說是沒有幫助的。如果在變更中進行了多個相關修改，請在列表中列出這些修改（同時仍保留資訊和小資訊）。描述是此更改的歷史記錄，程式碼搜尋等工具允許你查詢誰在程式碼庫中的任何特定更改中編寫了哪一行。在試圖修復 bug 時，深入瞭解原始更改通常很有用。

Descriptions aren't the only opportunity for adding documentation to a change. When writing a public API, you generally don't want to leak implementation details, but by all means do so within the actual implementation, where you should comment liberally. If a reviewer does not understand why you did something, even if it is correct, it is a good indicator that such code needs better structure or better comments (or both). If, during the code review process, a new decision is reached, update the change description, or add appropriate comments within the implementation. A code review is not just something that you do in the present time; it is something you do to record what you did for posterity.

描述並不是為一個更改新增文件的唯一機會。在編寫公共 API 時，你通常不想洩露實現的細節，但在實際實現中，你應該隨意地進行註釋。如果審查員不理解你為什麼要這樣做，即使它是正確的，這也是一個很好的指標，說明這樣的程式碼需要更好的結構或更好的註釋（或兩者）。如果在程式碼審查過程中，有了新的決定，請更新修改說明，或在實現中新增適當的註釋。程式碼審查不僅僅是你當前所做的事情；這是你為後繼者所做的記錄。

Keep Reviewers to a Minimum 儘量減少審查員

Most code reviews at Google are reviewed by precisely one reviewer.[9] Because the code review process allows the bits on code correctness, owner acceptance, and language readability to be handled by one individual, the code review process scales quite well across an organization the size of Google.

在谷歌，大多數的程式碼審查都是由一個審查員進行審查的。由於程式碼審查過程允許由一個人處理程式碼正確性、所有者接受度和語言可讀性等方面的問題，程式碼審查過程在谷歌這樣的組織規模中具有相當好的擴充性。

There is a tendency within the industry, and within individuals, to try to get additional input (and unanimous consent) from a cross-section of engineers. After all, each additional reviewer can add their own particular insight to the code review in question. But we've found that this leads to diminishing returns; the most important LGTM is the first one, and subsequent ones don't add as much as you might think to the equation. The cost of additional reviewers quickly outweighs their value.

在行業內和個人都有一種趨勢，即試圖從工程師的各個方面獲得額外的投入（和一致同意）。畢竟，每個額外的審查員都可以為所討論的程式碼審閱新增他們自己的特定見解。但我們發現這會導致收益遞減；最重要的 LGTM 是第一個，後續的 LGTM 不會像你想象的那樣新增到等式中。額外審查員的成本很快就超過了他們的價值。

The code review process is optimized around the trust we place in our engineers to do the right thing. In certain cases, it can be useful to get a particular change reviewed by multiple people, but even in those cases, those reviewers should focus on different aspects of the same change.

程式碼審查過程是圍繞著我們對工程師的信任而最佳化的，他們會做正確的事情。在某些情況下，讓一個特定的更改由多人審查可能是有用的，但即使在這些情況下，這些審查員也應該專注於同一變化的不同方面。

> 9 同上。

Automate Where Possible 儘可能實現自動化

Code review is a human process, and that human input is important, but if there are components of the code process that can be automated, try to do so. Opportunities to automate mechanical human tasks should be explored; investments in proper tooling reap dividends. At Google, our code review tooling allows authors to automatically submit and automatically sync changes to the source control system upon approval (usually used for fairly simple changes).

程式碼評審是一個人工過程，人的投入很重要，但是如果程式碼過程中的某些部分可以自動化，就儘量這樣做。應該探索將人類別的機械任務自動化的機會；在適當的工具上的投資可以獲得回報。在谷歌，我們的程式碼審查工具允許作者自動提交修改，並在批准後自動同步到原始碼控制系統（通常用於相當簡單的修改）。

One of the most important technological improvements regarding automation over the past few years is automatic static analysis of a given code change (see Chapter 20). Rather than require authors to run tests, linters, or formatters, the current Google code review tooling provides most of that utility automatically through what is known as *presubmits*. A presubmit process is run when a change is initially sent to a reviewer. Before that change is sent, the presubmit process can detect a variety of problems with the existing change, reject the current change (and prevent sending an awkward email to a reviewer), and ask the original author to fix the change first. Such automation not only helps out with the code review process itself, it also allows the reviewers to focus on more important concerns than formatting.

在過去的幾年中，關於自動化的最重要的技術改進之一是對給定的程式碼修改進行自動靜態分析（見第 20 章）。目前的 Google 程式碼審查工具並不要求作者執行測試、linters 或格式化程式，而是透過所謂的*預提*交自動提供大部分的效用。預提交的過程是在一個更改最初被髮送給一個審查員時執行的。在該更改被髮送之前，預提交程式可以檢測到現有更改的各種問題，拒絕當前的更改（並防止向審查者傳送尷尬的電子郵件），並要求原作者首先修復該更改。這樣的自動化不僅對程式碼審查過程本身有幫助，還能讓審查員專注於比格式化更重要的問題。

Types of Code Reviews 程式碼審查的型別

All code reviews are not alike! Different types of code review require different levels of focus on the various aspects of the review process. Code changes at Google generally fall into one of the following buckets (though there is sometimes overlap):

- Greenfield reviews and new feature development
- Behavioral changes, improvements, and optimizations
- Bug fixes and rollbacks
- Refactorings and large-scale changes

所有的程式碼審查都是不一樣的! 不同型別的程式碼審查需要對審查過程中的各個環節進行不同程度的關注。在谷歌，程式碼變更一般都屬於以下幾種情況（儘管有時會有重疊）：

- 綠地審查和新功能開發
- 行為更改、改進和最佳化
- bug 修復和回滾
- 重構和大規模更改

Greenfield Code Reviews 綠地程式碼審查

The least common type of code review is that of entirely new code, a so-called *green- field review*. A greenfield review is the most important time to evaluate whether the code will stand the test of time: that it will be easier to maintain as time and scale change the underlying assumptions of the code. Of course, the introduction of entirely new code should not come as a surprise. As mentioned earlier in this chapter, code is a liability, so the introduction of entirely new code should generally solve a real problem rather than simply provide yet another alternative. At Google, we generally require new code and/or projects to undergo an extensive design review, apart from a code review. A code review is not the time to debate design decisions already made in the past (and by the same token, a code review is not the time to introduce the design of a proposed API).

最不常見的程式碼審查型別是全新的程式碼，即所謂的*綠地審查*。綠地審查是評估程式碼是否經得起時間考驗的最重要時機：隨著時間和規模的變化，程式碼的基本假設也會發生變化，它將更容易維護。當然，引入全新的程式碼並不令人驚訝。正如本章前面提到的，編碼是一種責任，因此引入全新的程式碼通常應該解決一個真正的問題，而不僅僅是提供另一種選擇。在 Google，除了程式碼審查之外，我們一般要求新的程式碼和/或專案要經過廣泛的設計審查。程式碼審查不是辯論過去已經做出的設計決定的時候（同樣的道理，程式碼審查也不是介紹建議的 API 的設計的時候）。

To ensure that code is sustainable, a greenfield review should ensure that an API matches an agreed design (which may require reviewing a design document) and is tested *fully*, with all API endpoints having some form of unit test, and that those tests fail when the code's assumptions change. (See Chapter 11). The code should also have proper owners (one of the first reviews in a new project is often of a single OWNERS file for the new directory), be sufficiently commented, and provide supplemental documentation, if needed. A greenfield review might also necessitate the introduction of a project into the continuous integration system. (See Chapter 23).

為了確保程式碼是可持續性，綠地審查應該確保 API 與商定的設計相匹配（這可能需要審查設計文件），並進行*充分測試*，所有 API 端點都有某種形式的單元測試，而且當代碼的假設發生變化時，這些測試會失效。(見第 11 章）。程式碼還應該有適當的所有者（一個新專案的第一次審查往往是對新目錄的一個單一的 OWNERS 檔案的審查），有足夠的註釋，如果需要的話，還應該提供補充文件。綠地審查也可能需要將專案引入持續整合系統。(參見第 23 章）。

Behavioral Changes, Improvements, and Optimizations 行為更改、改進和最佳化

Most changes at Google generally fall into the broad category of modifications to existing code within the codebase. These additions may include modifications to API endpoints, improvements to existing implementations, or optimizations for other factors such as performance. Such changes are the bread and butter of most software engineers.

谷歌的大多數更改一般都屬於對程式碼庫內現有程式碼進行更改的型別。這些新增內容可能包括對 API 端點的更改，對現有實現的改進，或對其他因素的最佳化，如效能。這類更改是大多數軟體工程師的日常。

In each of these cases, the guidelines that apply to a greenfield review also apply: is this change necessary, and does this change improve the codebase? Some of the best modifications to a codebase are actually deletions! Getting rid of dead or obsolete code is one of the best ways to improve the overall code health of a codebase.

在每一種情況下，適用於綠地審查的指南也適用：這種更改是否必要，以及這種更改是否改善了程式碼庫？對程式碼庫的一些最佳更改實際上是刪除！消除死程式碼或過時程式碼是改善程式碼庫整體程式碼健康狀況的最佳方法之一。

Any behavioral modifications should necessarily include revisions to appropriate tests for any new API behavior. Augmentations to the implementation should be tested in a Continuous Integration (CI) system to ensure that those modifications don't break any underlying assumptions of the existing tests. As well, optimizations should of course ensure that they don't affect those tests and might need to include performance benchmarks for the reviewers to consult. Some optimizations might also require benchmark tests.

任何行為修改都必須包括對任何新 API 行為的適當測試的修訂。應在持續整合（CI）系統中測試對實現的增強，以確保這些修改不會破壞現有測試的任何基本假設。此外，最佳化當然應該確保它們不會影響這些測試，並且可能需要包括效能基準，供審查員參考。一些最佳化可能還需要基準測試。

Bug Fixes and Rollbacks Bug 修復和回滾

Inevitably, you will need to submit a change for a bug fix to your codebase. *When doing so, avoid the temptation to address other issues*. Not only does this risk increasing the size of the code review, it also makes it more difficult to perform regression testing or for others to roll back your change. A bug fix should focus solely on fixing the indicated bug and (usually) updating associated tests to catch the error that occurred in the first place.

不可避免地，你將需要提交一個更改以修復你的程式碼庫中的 bug。*在這樣做的時候，要避免一起處理其他問題的誘惑*。這不僅有可能增加程式碼審查的規模，也會使執行迴歸測試或其他人回滾你的更改更加困難。bug 修復應該只關注於修復指定的 bug，並且（通常）更新相關的測試以捕獲最初發生的錯誤。

Addressing the bug with a revised test is often necessary. The bug surfaced because existing tests were either inadequate, or the code had certain assumptions that were not met. As a reviewer of a bug fix, it is important to ask for updates to unit tests if applicable.

用修改後的測試來解決這個 bug 往往是必要的。這個 bug 的出現是因為現有的測試不充分，或者程式碼中的某些假設沒有被滿足。作為一個 bug 修復的審查員，如果適用的話，要求更新單元測試是很重要的。

Sometimes, a code change in a codebase as large as Google's causes some dependency to fail that was either not detected properly by tests or that unearths an untested part of the codebase. In those cases, Google allows such changes to be "rolled back," usually by the affected downstream customers. A rollback consists of a change that essentially undoes the previous change. Such rollbacks can be created in seconds because they just revert the previous change to a known state, but they still require a code review.

有時，像谷歌這樣龐大的程式碼庫中的程式碼變更會導致一些依賴失效，而這些依賴要麼沒有被測試正確地檢測到，要麼就是發現了程式碼庫中未經測試的部分。在這些情況下，谷歌允許這種變化被 "回滾"，通常是由受影響的下游客戶進行。回滾由基本上撤消先前更改的更改組成。這種回滾可以在幾秒鐘內建立，因為它們只是將以前的更改恢復到已知狀態，但仍然需要程式碼檢查。

It also becomes critically important that any change that could cause a potential rollback (and that includes all changes!) be as small and atomic as possible so that a rollback, if needed, does not cause further breakages on other dependencies that can be difficult to untangle. At Google, we've seen developers start to depend on new code very quickly after it is submitted, and rollbacks sometimes break these developers as a result. Small changes help to mitigate these concerns, both because of their atomicity, and because reviews of small changes tend to be done quickly.

同樣至關重要的是，任何可能導致潛在回滾的更改（這包括所有的變化！）都要儘可能小且原子化，這樣，如果需要回滾，就不會導致其他依賴關係的進一步破壞，從而難以解開。在谷歌，我們看到開發人員在提交新程式碼後很快就開始依賴新程式碼，而回滾有時會破壞這些開發人員。小更改有助於緩解這些擔憂，這既因為它們的原子性，也因為對小更改的審查往往很快完成。

Refactorings and Large-Scale Changes 重構和大規模更改

Many changes at Google are automatically generated: the author of the change isn't a person, but a machine. We discuss more about the large-scale change (LSC) process in Chapter 22, but even machine-generated changes require review. In cases where the change is considered low risk, it is reviewed by designated reviewers who have approval privileges for our entire codebase. But for cases in which the change might be risky or otherwise requires local domain expertise, individual engineers might be asked to review automatically generated changes as part of their normal workflow.

谷歌的許多變更是自動產生的：變更的作者不是人，而是機器。我們在第 22 章中討論了更多關於大規模變更(LSC)的過程，但即使是機器產生的變更也需要審查。在被認為是低風險的情況下，它被指定的審查員審查，他們對我們的整個程式碼庫有批准權。但對於那些可能有風險或需要本地領域專業知識的變化，個別工程師可能被要求審查自動產生的變化，作為他們正常工作流程的一部分。

At first look, a review for an automatically generated change should be handled the same as any other code review: the reviewer should check for correctness and applicability of the change. However, we encourage reviewers to limit comments in the associated change and only flag concerns that are specific to their code, not the underlying tool or LSC generating the changes. While the specific change might be machine generated, the overall process generating these changes has already been reviewed, and individual teams cannot hold a veto over the process, or it would not be possible to scale such changes across the organization. If there is a concern about the underlying tool or process, reviewers can escalate out of band to an LSC oversight group for more information.

乍一看，對自動產生的變更的審查應與任何其他程式碼審查一樣進行處理：審查員應檢查變更的正確性和適用性。但是，我們鼓勵審查員限制相關更改中的註釋，只標記特定於其程式碼的關注點，而不是產生更改的底層工具或 LSC。雖然具體的變更可能是機器產生的，但產生這些變更的整個流程已經過審查，單個團隊不能對該流程擁有否決權，否則就不可能在整個組織中擴充此類變更。如果對基礎工具或流程存在擔憂，審查員可以將帶外問題上報給 LSC 監督小組，以獲取更多資訊。

We also encourage reviewers of automatic changes to avoid expanding their scope. When reviewing a new feature or a change written by a teammate, it is often reasonable to ask the author to address related concerns within the same change, so long as the request still follows the earlier advice to keep the change small. This does not apply to automatically generated changes because the human running the tool might have hundreds of changes in flight, and even a small percentage of changes with review comments or unrelated questions limits the scale at which the human can effectively operate the tool.

我們還鼓勵自動更改的審查者避免擴大其範圍。當審查一項新功能或一項由團隊成員編寫的變更時，通常有理由要求作者解決同一變更中的相關問題，只要該請求仍然遵循先前的建議，以保持較小的變更。這不適用於自動產生的變更，因為執行工具的人可能有數百個變更在進行，即使是帶有審查意見或無關問題的一小部分更改，也會限制人員有效操作該工具的範圍。

Conclusion 總結

Code review is one of the most important and critical processes at Google. Code review acts as the glue connecting engineers with one another, and the code review process is the primary developer workflow upon which almost all other processes must hang, from testing to static analysis to CI. A code review process must scale appropriately, and for that reason, best practices, including small changes and rapid feedback and iteration, are important to maintain developer satisfaction and appropriate production velocity.

程式碼審查是谷歌最重要、最關鍵的流程之一。程式碼評審充當著工程師之間的粘合劑，程式碼評審過程是開發人員的主要工作流程，幾乎所有其他過程都必須依賴於此，從測試到靜態分析再到 CI。程式碼評審過程必須適當擴充，因此，最佳實踐（包括小變更、快速反饋和迭代）對於保持開發人員滿意度和適當的生產速度非常重要。

TL;DRs 內容提要

- Code review has many benefits, including ensuring code correctness, comprehension, and consistency across a codebase.
- Always check your assumptions through someone else; optimize for the reader.
- Provide the opportunity for critical feedback while remaining professional.
- Code review is important for knowledge sharing throughout an organization.

- Automation is critical for scaling the process.
- The code review itself provides a historical record.
- 程式碼審查有很多好處，包括確保程式碼的正確性、理解性和整個程式碼庫的一致性。
- 總是透過別人來檢查你的假設；為讀者最佳化。
- 在保持專業性同時提供關鍵反饋的機會。
- 程式碼審查對於整個組織的知識共享非常重要。
- 自動化對於擴充這個過程是至關重要的。
- 程式碼審查本身提供了一個歷史記錄。

---

1. We also use Gerrit to review Git code, primarily for our open source projects. However, Critique is the primary tool of a typical software engineer at Google. ↩

2. Steve McConnell, Code Complete (Redmond: Microsoft Press, 2004). ↩

3. At Google, “readability” does not refer simply to comprehension, but to the set of styles and best practices that allow code to be maintainable to other engineers. See Chapter 3. ↩

4. Some changes to documentation and configurations might not require a code review, but it is often still preferable to obtain such a review. ↩

5. “Advances in Software Inspection,” IEEE Transactions on Software Engineering, SE-12(7): 744–751, July 1986. Granted, this study took place before robust tooling and automated testing had become so important in the software development process, but the results still seem relevant in the modern software age. ↩

6. Rigby, Peter C. and Christian Bird. 2013. “Convergent software peer review practices.” ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, August 2013: 202-212. https:// dl.acm.org/doi/10.1145/2491411.2491444. ↩

7. Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli, “Modern code review: a case study at Google.” ↩

8. Ibid. ↩

9. Ibid. ↩