Chapter 15. Deprecation

Written by Hyrum Wright

Edited by Tom Manshreck

> *I love deadlines. I like the whooshing sound they make as they fly by.*
>
> —Douglas Adams

All systems age. Even though software is a digital asset and the physical bits themselves don't degrade, new technologies, libraries, techniques, languages, and other environmental changes over time render existing systems obsolete. Old systems require continued maintenance, esoteric expertise, and generally more work as they diverge from the surrounding ecosystem. It's often better to invest effort in turning off obsolete systems, rather than letting them lumber along indefinitely alongside the systems that replace them. But the number of obsolete systems still running suggests that, in practice, doing so is not trivial. We refer to the process of orderly migration away from and eventual removal of obsolete systems as *deprecation*.

Deprecation is yet another topic that more accurately belongs to the discipline of software engineering than programming because it requires thinking about how to manage a system over time. For long-running software ecosystems, planning for and executing deprecation correctly reduces resource costs and improves velocity by removing the redundancy and complexity that builds up in a system over time. On the other hand, poorly deprecated systems may cost more than leaving them alone. While deprecating systems requires additional effort, it's possible to plan for deprecation during the design of the system so that it's easier to eventually decommission and remove it. Deprecations can affect systems ranging from individual function calls to entire software stacks. For concreteness, much of what follows focuses on code-level deprecations.

Unlike with most of the other topics we have discussed in this book, Google is still learning how best to deprecate and remove software systems. This chapter describes the lessons we've learned as we've deprecated large and heavily

used internal systems. Sometimes, it works as expected, and sometimes it doesn't, but the general problem of removing obsolete systems remains a difficult and evolving concern in the industry.

This chapter primarily deals with deprecating technical systems, not end-user products. The distinction is somewhat arbitrary given that an external-facing API is just another sort of product, and an internal API may have consumers that consider themselves end users. Although many of the principles apply to turning down a public product, we concern ourselves here with the technical and policy aspects of deprecating and removing obsolete systems where the system owner has visibility into its use.

Why Deprecate?

Our discussion of deprecation begins from the fundamental premise that *code is a liability, not an asset*. After all, if code were an asset, why should we even bother spending time trying to turn down and remove obsolete systems? Code has costs, some of which are borne in the process of creating a system, but many other costs are borne as a system is maintained across its lifetime. These ongoing costs, such as the operational resources required to keep a system running or the effort to continually update its codebase as surrounding ecosystems evolve, mean that it's worth evaluating the trade-offs between keeping an aging system running or working to turn it down.

The age of a system alone doesn't justify its deprecation. A system could be finely crafted over several years to be the epitome of software form and function. Some software systems, such as the LaTeX typesetting system, have been improved over the course of decades, and even though changes still happen, they are few and far between. Just because something is old, it does not follow that it is obsolete.

Deprecation is best suited for systems that are demonstrably obsolete and a replacement exists that provides comparable functionality. The new system might use resources more efficiently, have better security properties, be built in a more sustainable fashion, or just fix bugs. Having two systems to accomplish the same thing might not seem like a pressing problem, but over time, the costs of maintaining them both can grow substantially. Users may need to use the new system, but still have dependencies that use the obsolete one.

The two systems might need to interface with each other, requiring complicated transformation code. As both systems evolve, they may come to depend on each other, making eventual removal of either more difficult. In the long run, we've discovered that having multiple systems performing the same function also impedes the evolution of the newer system because it is still expected to maintain compatibility with the old one. Spending the effort to remove the old system can pay off as the replacement system can now evolve more quickly.

Earlier we made the assertion that "code is a liability, not an asset." If that is true, why have we spent most of this book discussing the most efficient way to build software systems that can live for decades? Why put all that effort into creating more code when it's simply going to end up on the liability side of the balance sheet?

Code *itself* doesn't bring value: it is the *functionality* that it provides that brings value. That functionality is an asset if it meets a user need: the code that implements this functionality is simply a means to that end. If we could get the same functionality from a single line of maintainable, understandable code as 10,000 lines of convoluted spaghetti code, we would prefer the former. Code itself carries a cost—the simpler the code is, while maintaining the same amount of functionality, the better.

Instead of focusing on how much code we can produce, or how large is our codebase, we should instead focus on how much functionality it can deliver per unit of code and try to maximize that metric. One of the easiest ways to do so isn't writing more code and hoping to get more functionality; it's removing excess code and systems that are no longer needed. Deprecation policies and procedures make this possible.

Even though deprecation is useful, we've learned at Google that organizations have a limit on the amount of deprecation work that is reasonable to undergo simultaneously, from the aspect of the teams doing the deprecation as well as the customers of those teams. For example, although everybody appreciates having freshly paved roads, if the public works department decided to close down *every* road for paving simultaneously, nobody would go anywhere. By focusing their efforts, paving crews can get specific jobs done faster while also allowing other traffic to make progress. Likewise, it's important to

choose deprecation projects with care and then commit to following through on finishing them.

Why Is Deprecation So Hard?

We've mentioned Hyrum's Law elsewhere in this book, but it's worth repeating its applicability here: the more users of a system, the higher the probability that users are using it in unexpected and unforeseen ways, and the harder it will be to deprecate and remove such a system. Their usage just "happens to work" instead of being "guaranteed to work." In this context, removing a system can be thought of as the ultimate change: we aren't just changing behavior, we are removing that behavior completely! This kind of radical alteration will shake loose a number of unexpected dependents.

To further complicate matters, deprecation usually isn't an option until a newer system is available that provides the same (or better!) functionality. The new system might be better, but it is also different: after all, if it were exactly the same as the obsolete system, it wouldn't provide any benefit to users who migrate to it (though it might benefit the team operating it). This functional difference means a one-to-one match between the old system and the new system is rare, and every use of the old system must be evaluated in the context of the new one.

Another surprising reluctance to deprecate is emotional attachment to old systems, particularly those that the deprecator had a hand in helping to create. An example of this change aversion happens when systematically removing old code at Google: we've occasionally encountered resistance of the form "I like this code!" It can be difficult to convince engineers to tear down something they've spent years building. This is an understandable response, but ultimately self-defeating: if a system is obsolete, it has a net cost on the organization and should be removed. One of the ways we've addressed concerns about keeping old code within Google is by ensuring that the source code repository isn't just searchable at trunk, but also historically. Even code that has been removed can be found again (see Chapter 17).

There's an old joke within Google that there are two ways of doing things: the one that's deprecated, and the one that's not-yet-ready. This is usually the result of a new solution being "almost" done and is the unfortunate reality of working in a technological environment that is complex and fast-paced.

Google engineers have become used to working in this environment, but it can still be disconcerting. Good documentation, plenty of signposts, and teams of experts helping with the deprecation and migration process all make it easier to know whether you should be using the old thing, with all its warts, or the new one, with all its uncertainties.

Finally, funding and executing deprecation efforts can be difficult politically; staffing a team and spending time removing obsolete systems costs real money, whereas the costs of doing nothing and letting the system lumber along unattended are not readily observable. It can be difficult to convince the relevant stakeholders that deprecation efforts are worthwhile, particularly if they negatively impact new feature development. Research techniques, such as those described in Chapter 7, can provide concrete evidence that a deprecation is worthwhile.

Given the difficulty in deprecating and removing obsolete software systems, it is often easier for users to evolve a system *in situ*, rather than completely replacing it. Incrementality doesn't avoid the deprecation process altogether, but it does break it down into smaller, more manageable chunks that can yield incremental benefits. Within Google, we've observed that migrating to entirely new systems is *extremely* expensive, and the costs are frequently underestimated. Incremental deprecation efforts accomplished by in-place refactoring can keep existing systems running while making it easier to deliver value to users.

Deprecation During Design

Like many engineering activities, deprecation of a software system can be planned as those systems are first built. Choices of programming language, software architecture, team composition, and even company policy and culture all impact how easy it will be to eventually remove a system after it has reached the end of its useful life.

The concept of designing systems so that they can eventually be deprecated might be radical in software engineering, but it is common in other engineering disciplines. Consider the example of a nuclear power plant, which is an extremely complex piece of engineering. As part of the design of a nuclear power station, its eventual decommissioning after a lifetime of productive service must be taken into account, even going so far as to allocate funds for this purpose.[1] Many of the design choices in building a nuclear power plant are affected when engineers know that it will eventually need to be decommissioned.

Unfortunately, software systems are rarely so thoughtfully designed. Many software engineers are attracted to the task of building and launching new systems, not maintaining existing ones. The corporate culture of many companies, including Google, emphasizes building and shipping new products quickly, which often provides a disincentive for designing with deprecation in mind from the beginning. And in spite of the popular notion of software engineers as data-driven automata, it can be psychologically difficult to plan for the eventual demise of the creations we are working so hard to build.

So, what kinds of considerations should we think about when designing systems that we can more easily deprecate in the future? Here are a couple of the questions we encourage engineering teams at Google to ask:

- How easy will it be for my consumers to migrate from my product to a potential replacement?
- How can I replace parts of my system incrementally?

Many of these questions relate to how a system provides and consumes dependencies. For a more thorough discussion of how we manage these dependencies, see Chapter 16.

Finally, we should point out that the decision as to whether to support a project long term is made when an organization first decides to build the project. After a software system exists, the only remaining options are support it, carefully deprecate it, or let it stop functioning when some external event causes it to break. These are all valid options, and the trade-offs between them will be organization specific. A new startup with a single project will unceremoniously kill it when the company goes bankrupt, but a large company will need to think more closely about the impact across its portfolio and reputation as they consider removing old projects. As mentioned earlier,

Google is still learning how best to make these trade-offs with our own internal and external products.

In short, don't start projects that your organization isn't committed to support for the expected lifespan of the organization. Even if the organization chooses to deprecate and remove the project, there will still be costs, but they can be mitigated through planning and investments in tools and policy.

Types of Deprecation

Deprecation isn't a single kind of process, but a continuum of them, ranging from "we'll turn this off someday, we hope" to "this system is going away tomorrow, customers better be ready for that." Broadly speaking, we divide this continuum into two separate areas: advisory and compulsory.

Advisory Deprecation

Advisory deprecations are those that don't have a deadline and aren't high priority for the organization (and for which the company isn't willing to dedicate resources). These could also be labeled *aspirational* deprecations: the team knows the system has been replaced, and although they hope clients will eventually migrate to the new system, they don't have imminent plans to either provide support to help move clients or delete the old system. This kind of deprecation often lacks enforcement: we hope that clients move, but can't force them to. As our friends in SRE will readily tell you: "Hope is not a strategy."

Advisory deprecations are a good tool for advertising the existence of a new system and encouraging early adopting users to start trying it out. Such a new system should *not* be considered in a beta period: it should be ready for production uses and loads and should be prepared to support new users indefinitely. Of course, any new system is going to experience growing pains, but after the old system has been deprecated in any way, the new system will become a critical piece of the organization's infrastructure.

One scenario we've seen at Google in which advisory deprecations have strong benefits is when the new system offers compelling benefits to its users. In these cases, simply notifying users of this new system and providing them self-service tools to migrate to it often encourages adoption. However, the

benefits cannot be simply incremental: they must be transformative. Users will be hesitant to migrate on their own for marginal benefits, and even new systems with vast improvements will not gain full adoption using only advisory deprecation efforts.

Advisory deprecation allows system authors to nudge users in the desired direction, but they should not be counted on to do the majority of migration work. It is often tempting to simply put a deprecation warning on an old system and walk away without any further effort. Our experience at Google has been that this can lead to (slightly) fewer new uses of an obsolete system, but it rarely leads to teams actively migrating away from it. Existing uses of the old system exert a sort of conceptual (or technical) pull toward it: comparatively many uses of the old system will tend to pick up a large share of new uses, no matter how much we say, "Please use the new system." The old system will continue to require maintenance and other resources unless its users are more actively encouraged to migrate.

Compulsory Deprecation

This active encouragement comes in the form of *compulsory* deprecation. This kind of deprecation usually comes with a deadline for removal of the obsolete system: if users continue to depend on it beyond that date, they will find their own systems no longer work.

Counterintuitively, the best way for compulsory deprecation efforts to scale is by localizing the expertise of migrating users to within a single team of experts—usually the team responsible for removing the old system entirely. This team has incentives to help others migrate from the obsolete system and can develop experience and tools that can then be used across the organization. Many of these migrations can be effected using the same tools discussed in Chapter 22.

For compulsory deprecation to actually work, its schedule needs to have an enforcement mechanism. This does not imply that the schedule can't change, but empower the team running the deprecation process to break noncompliant users after they have been sufficiently warned through efforts to migrate them. Without this power, it becomes easy for customer teams to ignore deprecation work in favor of features or other more pressing work.

At the same time, compulsory deprecations without staffing to do the work can come across to customer teams as mean spirited, which usually impedes completing the deprecation. Customers simply see such deprecation work as an unfunded mandate, requiring them to push aside their own priorities to do work just to keep their services running. This feels much like the "running to stay in place" phenomenon and creates friction between infrastructure maintainers and their customers. It's for this reason that we strongly advocate that compulsory deprecations are actively staffed by a specialized team through completion.

It's also worth noting that even with the force of policy behind them, compulsory deprecations can still face political hurdles. Imagine trying to enforce a compulsory deprecation effort when the last remaining user of the old system is a critical piece of infrastructure your entire organization depends on. How willing would you be to break that infrastructure—and, transitively, everybody that depends on it—just for the sake of making an arbitrary deadline? It is hard to believe the deprecation is really compulsory if that team can veto its progress.

Google's monolithic repository and dependency graph gives us tremendous insight into how systems are used across our ecosystem. Even so, some teams might not even know they have a dependency on an obsolete system, and it can be difficult to discover these dependencies analytically. It's also possible to find them dynamically through tests of increasing frequency and duration during which the old system is turned off temporarily. These intentional changes provide a mechanism for discovering unintended dependencies by seeing what breaks, thus alerting teams to a need to prepare for the upcoming deadline. Within Google, we occasionally change the name of implementation-only symbols to see which users are depending on them unaware.

Frequently at Google, when a system is slated for deprecation and removal, the team will announce planned outages of increasing duration in the months and weeks prior to the turndown. Similar to Google's Disaster Recovery Testing (DiRT) exercises, these events often discover unknown dependencies between running systems. This incremental approach allows those dependent teams to discover and then plan for the system's eventual removal, or even work with the deprecating team to adjust their timeline. (The same principles also apply for static code dependencies, but the semantic information provided

by static analysis tools is often sufficient to detect all the dependencies of the obsolete system.)

Deprecation Warnings

For both advisory and compulsory deprecations, it is often useful to have a programmatic way of marking systems as deprecated so that users are warned about their use and encouraged to move away. It's often tempting to just mark something as deprecated and hope its uses eventually disappear, but remember: "hope is not a strategy." Deprecation warnings can help prevent new uses, but rarely lead to migration of existing systems.

What usually happens in practice is that these warnings accumulate over time. If they are used in a transitive context (for example, library A depends on library B, which depends on library C, and C issues a warning, which shows up when A is built), these warnings can soon overwhelm users of a system to the point where they ignore them altogether. In health care, this phenomenon is known as "alert fatigue."

Any deprecation warning issued to a user needs to have two properties: actionability and relevance. A warning is *actionable* if the user can use the warning to actually perform some relevant action, not just in theory, but in practical terms, given the expertise in that problem area that we expect for an average engineer. For example, a tool might warn that a call to a given function should be replaced with a call to its updated counterpart, or an email might outline the steps required to move data from an old system to a new one. In each case, the warning provided the next steps that an engineer can perform to no longer depend on the deprecated system.[2]

A warning can be actionable, but still be annoying. To be useful, a deprecation warning should also be *relevant*. A warning is relevant if it surfaces at a time when a user actually performs the indicated action. Warning about the use of a deprecated function is best done while the engineer is writing code that uses that function, not after it has been checked into the repository for several weeks. Likewise, an email for data migration is best sent several months before the old system is removed rather than as an afterthought a weekend before the removal occurs.

It's important to resist the urge to put deprecation warnings on everything possible. Warnings themselves are not bad, but naive tooling often produces a

quantity of warning messages that can overwhelm the unsuspecting engineer. Within Google, we are very liberal with marking old functions as deprecated but leverage tooling such as ErrorProne or clang-tidy to ensure that warnings are surfaced in targeted ways. As discussed in Chapter 20, we limit these warnings to newly changed lines as a way to warn people about new uses of the deprecated symbol. Much more intrusive warnings, such as for deprecated targets in the dependency graph, are added only for compulsory deprecations, and the team is actively moving users away. In either case, tooling plays an important role in surfacing the appropriate information to the appropriate people at the proper time, allowing more warnings to be added without fatiguing the user.

Managing the Deprecation Process

Although they can feel like different kinds of projects because we're deconstructing a system rather than building it, deprecation projects are similar to other software engineering projects in the way they are managed and run. We won't spend too much effort going over similarities between those management efforts, but it's worth pointing out the ways in which they differ.

Process Owners

We've learned at Google that without explicit owners, a deprecation process is unlikely to make meaningful progress, no matter how many warnings and alerts a system might generate. Having explicit project owners who are tasked with managing and running the deprecation process might seem like a poor use of resources, but the alternatives are even worse: don't ever deprecate anything, or delegate deprecation efforts to the users of the system. The second case becomes simply an advisory deprecation, which will never organically finish, and the first is a commitment to maintain every old system ad infinitum. Centralizing deprecation efforts helps better assure that expertise actually *reduces* costs by making them more transparent.

Abandoned projects often present a problem when establishing ownership and aligning incentives. Every organization of reasonable size has projects that are still actively used but that nobody clearly owns or maintains, and Google is no exception. Projects sometimes enter this state because they are deprecated: the original owners have moved on to a successor project, leaving the obsolete

one chugging along in the basement, still a dependency of a critical project, and hoping it just fades away eventually.

Such projects are unlikely to fade away on their own. In spite of our best hopes, we've found that these projects still require deprecation experts to remove them and prevent their failure at inopportune times. These teams should have removal as their primary goal, not just a side project of some other work. In the case of competing priorities, deprecation work will almost always be perceived as having a lower priority and rarely receive the attention it needs. These sorts of important-not-urgent cleanup tasks are a great use of 20% time and provide engineers exposure to other parts of the codebase.

Milestones

When building a new system, project milestones are generally pretty clear: "Launch the frobnazzer features by next quarter." Following incremental development practices, teams build and deliver functionality incrementally to users, who get a win whenever they take advantage of a new feature. The end goal might be to launch the entire system, but incremental milestones help give the team a sense of progress and ensure they don't need to wait until the end of the process to generate value for the organization.

In contrast, it can often feel that the only milestone of a deprecation process is removing the obsolete system entirely. The team can feel they haven't made any progress until they've turned out the lights and gone home. Although this might be the most meaningful step for the team, if it has done its job correctly, it's often the least noticed by anyone external to the team, because by that point, the obsolete system no longer has any users. Deprecation project managers should resist the temptation to make this the only measurable milestone, particularly given that it might not even happen in all deprecation projects.

Similar to building a new system, managing a team working on deprecation should involve concrete incremental milestones, which are measurable and deliver value to users. The metrics used to evaluate the progress of the deprecation will be different, but it is still good for morale to celebrate incremental achievements in the deprecation process. We have found it useful to recognize appropriate incremental milestones, such as deleting a key subcomponent, just as we'd recognize accomplishments in building a new product.

Deprecation Tooling

Much of the tooling used to manage the deprecation process is discussed in depth elsewhere in this book, such as the large-scale change (LSC) process (Chapter 22) or our code review tools (Chapter 19). Rather than talk about the specifics of the tools, we'll briefly outline how those tools are useful when managing the deprecation of an obsolete system. These tools can be categorized as discovery, migration, and backsliding prevention tooling.

Discovery

During the early stages of a deprecation process, and in fact during the entire process, it is useful to know *how* and *by whom* an obsolete system is being used. Much of the initial work of deprecation is determining who is using the old system—and in which unanticipated ways. Depending on the kinds of use, this process may require revisiting the deprecation decision once new information is learned. We also use these tools throughout the deprecation process to understand how the effort is progressing.

Within Google, we use tools like Code Search (see Chapter 17) and Kythe (see Chapter 23) to statically determine which customers use a given library, and often to sample existing usage to see what sorts of behaviors customers are unexpectedly depending on. Because runtime dependencies generally require some static library or thin client use, this technique yields much of the information needed to start and run a deprecation process. Logging and runtime sampling in production help discover issues with dynamic dependencies.

Finally, we treat our global test suite as an oracle to determine whether all references to an old symbol have been removed. As discussed in Chapter 11, tests are a mechanism of preventing unwanted behavioral changes to a system as the ecosystem evolves. Deprecation is a large part of that evolution, and customers are responsible for having sufficient testing to ensure that the removal of an obsolete system will not harm them.

Migration

Much of the work of doing deprecation efforts at Google is achieved by using the same set of code generation and review tooling we mentioned earlier. The

LSC process and tooling are particularly useful in managing the large effort of actually updating the codebase to refer to new libraries or runtime services.

Preventing backsliding

Finally, an often overlooked piece of deprecation infrastructure is tooling for preventing the addition of new uses of the very thing being actively removed. Even for advisory deprecations, it is useful to warn users to shy away from a deprecated system in favor of a new one when they are writing new code. Without backsliding prevention, deprecation can become a game of whack-a-mole in which users constantly add new uses of a system with which they are familiar (or find examples of elsewhere in the codebase), and the deprecation team constantly migrates these new uses. This process is both counterproductive and demoralizing.

To prevent deprecation backsliding on a micro level, we use the Tricorder static analysis framework to notify users that they are adding calls into a deprecated system and give them feedback on the appropriate replacement. Owners of deprecated systems can add compiler annotations to deprecated symbols (such as the `@deprecated` Java annotation), and Tricorder surfaces new uses of these symbols at review time. These annotations give control over messaging to the teams that own the deprecated system, while at the same time automatically alerting the change author. In limited cases, the tooling also suggests a push-button fix to migrate to the suggested replacement.

On a macro level, we use visibility whitelists in our build system to ensure that new dependencies are not introduced to the deprecated system. Automated tooling periodically examines these whitelists and prunes them as dependent systems are migrated away from the obsolete system.

Conclusion

Deprecation can feel like the dirty work of cleaning up the street after the circus parade has just passed through town, yet these efforts improve the overall software ecosystem by reducing maintenance overhead and cognitive burden of engineers. Scalably maintaining complex software systems over time is more than just building and running software: we must also be able to remove systems that are obsolete or otherwise unused.

A complete deprecation process involves successfully managing social and technical challenges through policy and tooling. Deprecating in an organized and well-managed fashion is often overlooked as a source of benefit to an organization, but is essential for its long-term sustainability.

TL;DRs

- Software systems have continuing maintenance costs that should be weighed against the costs of removing them.
- Removing things is often more difficult than building them to begin with because existing users are often using the system beyond its original design.
- Evolving a system in place is usually cheaper than replacing it with a new one, when turndown costs are included.
- It is difficult to honestly evaluate the costs involved in deciding whether to deprecate: aside from the direct maintenance costs involved in keeping the old system around, there are ecosystem costs involved in having multiple similar systems to choose between and that might need to interoperate. The old system might implicitly be a drag on feature development for the new. These ecosystem costs are diffuse and difficult to measure. Deprecation and removal costs are often similarly diffuse.

1 "Design and Construction of Nuclear Power Plants to Facilitate Decommissioning," Technical Reports Series No. 382, IAEA, Vienna (1997).

2 See *https://abseil.io/docs/cpp/tools/api-upgrades* for an example.