# Release Early, Release Often

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release a version every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not—because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle [[QR].](http://www.catb.org/%7Eesr/writings/cathedral-bazaar/cathedral-bazaar/ar01s14.html#QR)

The most important of these, the Ohio State Emacs Lisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in the FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

7. Release early. Release often. And listen to your customers.

Linus's innovation wasn't so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a day! Because he cultivated his base of codevelopers and leveraged the Internet for collaboration harder than anyone else, this worked.

But how did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I didn't think so. Granted, Linus is a damn fine hacker. How many of us could engineer an entire productionquality operating system kernel from scratch? But Linux didn't represent any awesome conceptual leap forward. Linus is not (or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering and implementation, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded—stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even daily) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Or, less formally, ``Given enough eyeballs, all bugs are shallow.'' I dub this: ``Linus's Law''.

My original formulation was that every problem ``will be transparent to somebody''. Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. ``Somebody finds the problem,'' he says, ``and somebody else understands it. And I'll go on record as saying that finding it is the bigger challenge.'' That correction is important; we'll see how in the next section, when we examine the practice of debugging in more detail. But the key point is that both parts of the process (finding and fixing) tend to happen rapidly.

In Linus's Law, I think, lies the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena—or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that's it. That's enough. If ``Linus's Law'' is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the that kernel was, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered ``deep'' bugs. If it's true, on the other hand, it is sufficient to explain Linux's relative lack of bugginess and its continuous uptimes spanning months or even years.

Maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than the opinion of a single randomly-chosen one of the observers. They called this the Delphi effect. It appears that what Linus has shown is that this applies even to debugging an operating system—that the Delphi effect can tame development complexity even at the complexity level of an OS kernel. [[CV]](http://www.catb.org/%7Eesr/writings/cathedral-bazaar/cathedral-bazaar/ar01s14.html#CV)

One special feature of the Linux situation that clearly helps along the Delphi effect is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

Linus's Law can be rephrased as ``Debugging is parallelizable''. Although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a ``release early and often'' policy is to minimize such duplication by propagating fed-back fixes quickly [[JH].](http://www.catb.org/%7Eesr/writings/cathedral-bazaar/cathedral-bazaar/ar01s14.html#JH)

Brooks (the author of The Mythical Man-Month) even made an off-hand observation related to this: ``The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. More users find more bugs.'' [emphasis added].

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The ``Delphi effect'' seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

So adding more beta-testers may not reduce the complexity of the current ``deepest'' bug from the developer's point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow to that person.

Linus coppers his bets, too. In case there are serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated ``stable'' or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet systematically imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive. [[HBS]](http://www.catb.org/%7Eesr/writings/cathedral-bazaar/cathedral-bazaar/ar01s14.html#HBS)